SRI International

NASA-CR-192397

RESEARCH ON PROCEDURAL REASONING SYSTEMS

[illegible]

FINAL REPORT — PHASE 1
Covering Period - October 31, 1986 to October 30, 1988

October 1988

By: Michael P. Georgeff, Senior Computer Scientist
François Felix Ingrand, Computer Scientist
Representation and Reasoning Program
Artificial Intelligence Center
Computer and Information Sciences Division

Prepared for

NASA/Ames Research Center
Moffett Field, California
Attention: Mr. Don McKellar

Contract No. NAS2-12521
SRI Project 2851

APPROVED:

Oscar Firschein, Acting Director
Representation and Reasoning Program

Raymond Perrault, Director
Artificial Intelligence Center

SRI International®
333 Ravenswood Ave. • Menlo Park, CA 94025
415 326-6200 • TWX: 910-373-2046 • Telex: 334-486

Contents

Chapter 1

Introduction

This report describes work concerned with automating the operation of complex physical systems; in particular, those involved in space missions. Such operations include subsystem monitoring, preventative maintenance, malfunction handling, fault isolation and diagnosis, communications management, maintenance of life support systems, power management, monitoring of experiments, satellite servicing, payload deployment, orbital-vehicle operations, orbital construction and assembly, and control of extraterrestrial rovers. Automation of these tasks can be expected to improve mission productivity and safety, increase versatility, lessen dependence on ground systems, and reduce demands for crew involvement in system operations.

1.1 Design of an Embedded Reasoning System

The aim of our work is to design an embedded reasoning system capable of assisting astronauts in handling tasks such as those mentioned above. The system should be capable of responding to and diagnosing abnormalities in space vehicle operations. It should be able to integrate information from various parts of the space vehicle systems, and recognize potential problems prior to alarm limits being exceeded.

The system should suggest and execute strategies for containing damage and for making the system secure, without losing critical diagnostic information. It should be able to utilize standard malfunction handling procedures, taking account of all the relevant factors that, in crisis situations, are easily overlooked. False alarms and invalid parameter readings should be detected, and alternative means for deducing parameter values utilized where possible.

In parallel with efforts to contain damage and temporarily reconfigure vehicle subsystems, the system should be able to begin diagnosis of the problem and adjust reconfiguration strategies as diagnostic information is obtained incrementally. The system should also be capable of communicating with other systems to seek information, advise of critical conditions, and avoid harmful interactions. Throughout this process, the system should be continually reevaluating the state of the space vehicle, and be capable of changing focus and responding to more serious problems should they occur.

Finally, the system should be able to explain the reasons for any proposed course of action in terms that are familiar to astronauts and mission controllers. It should be able to graphically display its schematics, the procedures it is intending to execute, and the critical parameter values upon which its judgement is based.

This view of the system, together with the strategic and operational requirements of NASA's space program, demands that the system be designed to

- Monitor and react in real time to the changes in situation that arise during space operations.
- Rapidly select and execute the most appropriate operational procedures and strategies in any given situation.
- Represent and reason about critical properties that determine operational options, such as the present configuration or mode of the spacecraft and the relevant flight rules.
- Be robust, i.e., cope with modifications to planned goals and priorities, and not fail in the presence of ambiguous or inaccurate feedback from the operational environment.
- Support distributed operations, i.e., interact and cooperate with other systems that are monitoring and controlling other parts of the spacecraft.
- Interact with the astronauts and mission controllers in a clearly understandable way, i.e., appear to behave as much like a rational human assistant as possible.
- Allow established operational procedures and strategies to be changed easily without extensive modifications of source code when better strategies are developed.
- Be extensible, i.e., allow changes in the configuration or design of the spacecraft and its subsystems to be incorporated easily.
- Be verifiable, i.e., provide a clear meaning for the knowledge represented in the system and to ensure that the system behaves correctly with respect to this knowledge.

1.2 Previous Approaches

Previous attempts to design embedded reasoning systems have been able to accomplish few of the goals mentioned above. These approaches are discussed briefly in the following sections.

1.2.1 Conventional Software

It is highly unlikely that systems based on conventional automation techniques would be able to achieve the performance criteria discussed above. Such systems are usually very inflexible and unresponsive to the skill level of the operators using them. The control and diagnostic procedures that are used cannot be matched to the exigencies of the current situation nor can they cope with reconfiguration or modification of the underlying space-vehicle systems. Performance of tasks cannot be guided by useful advice from astronauts and technicians and, when a given task cannot be performed, no explanation is given as to the cause of failure. Because conventional systems perform a prescribed sequence of tests and actions, they cannot utilize knowledge of a particular situation to focus attention on more likely trouble spots. Consequently, real-time performance is highly unsatisfactory. They lack robustness, as they cannot fall back on other, perhaps less optimal, operational procedures if the current one fails to achieve its purpose. Furthermore, it is difficult and costly to modify or improve the encoded operational and malfunction-handling strategies and procedures.

1.2.2 Conventional Expert Systems

Conventional expert systems are not designed to handle effectively the reasoning and planning that must be performed under dynamic, real-time constraints. Unless one is concerned solely with simple monitoring tasks, it is essential that any embedded reasoning system be able to reason about actions and plans. Most existing real-time expert systems provide means for the system to interact with its environment through sensors and effectors, but their representations and reasoning capabilities differ little from those of conventional expert systems. These systems are far too weak to be used effectively for the kind of complex operations with which we are concerned. One must take the notions of actions and plans seriously, and utilize representations and reasoning mechanisms suited to handling these entities. A review of existing real-time reasoning systems and their shortcomings can be found in reference [10].

Another major obstacle to the use of conventional rule-based expert systems is that space operations require the use of a great variety of operational procedures. Conventional expert systems are simply not designed to represent or utilize such procedural knowledge. The various tests and actions performed on a spacecraft have diverse outcomes with different implications in different contexts. The only way to represent this in a rule-based formalism is to keep track of the procedural context by the use of so-called *control conditions*. This form of representation becomes very clumsy, reduces efficiency, and nullifies most of the desirable properties of an expert system. In essence, the rule-based approach makes things implicit that should be explicit (i.e., the flow of control) and makes things explicit that should be implicit (i.e., the context).

With the addition of the control conditions necessary to represent procedural information, extensibility and robustness are lost; each control condition must be unique and should not be used by any other rule other than the one for which it was intended. Explanatory capacity is poor, as there is no direct access to the entire procedure; each rule must be explicated in isolation – with no satisfactory explanation for the meaning or use of the control conditions. Moreover, the validity of a rule containing a control condition depends on the validity of the rule or rules that inserted the control condition into the database, which in turn depend on the rules that inserted their control conditions into the database, and so on. One could never be certain that a rule would not be invoked unexpectedly, with perhaps catastrophic effects. Furthermore, and perhaps most importantly, it is not possible for the system to reason about a procedure as a whole – for example, to assess its usefulness or criticality in a given situation.

Experience in trying to apply conventional expert systems to problems in fault diagnosis and maintenance has shown that these difficulties are severe. To overcome them, some expert systems offer facilities for representing procedural knowledge (e.g., Centaur [1]). In most cases, however, such procedures are represented simply by LISP code (or some equivalent) that can be invoked via the database. The procedures are *ad hoc* additions, have limited control constructs, cannot be reasoned about, and cannot be interrupted on the basis of newly observed data or newly established goals.[1]

1.2.3 Planning Systems

Most existing architectures for embedded planning systems consist of a plan constructor and a plan executor. As a rule, the plan constructor formulates an entire course of action before commencing execution of the plan [14]. Execution is usually monitored to ensure that the plan will culminate in the desired effects; if it does not, the system can return control to the plan constructor so that it may modify the existing plan appropriately.

However, in real-world domains, much of the information about how best to achieve a given goal is acquired during plan execution. For example, the choice of how to best normalize tank pressure while handling a jet failure may depend on observations made during the diagnostic process. In such situations, one cannot use a system that plans in full prior to commencing execution.

Real-time constraints pose yet further problems for traditionally structured planning and reasoning systems. First, the planning and deductive techniques typically used by these systems are very time consuming. While this may be acceptable in some situations, it is not suited to domains where replanning is frequently necessary and where system viability depends on readiness to act. Second, traditional planning systems usually provide

[1]The impracticality of utilizing standard expert systems to represent procedural knowledge is discussed in greater detail in reference [8].

no mechanisms for responding to new situations or goals during plan execution, let alone during plan formation. Yet the very survival of an autonomous system may depend on its ability to react to new situations and to modify its goals and intentions accordingly. For example, a space-station robot should be capable of deferring work on an onboard experiment if it notices something more critical, such as a possible jet malfunction. While many existing planners have replanning capabilities, none have yet accommodated modifications to the system's underlying set of goal priorities.

Furthermore, traditional planning systems have been designed for constructing plans solely from knowledge about the primitive actions performable by the system. However, many plans are not constructed from first principles, but from knowledge acquired in a variety of other ways — for example, by being told, by learning, or through training. Moreover, these plans may be very complex, involving a variety of control constructs (such as iteration and recursion) that are normally not part of the repertoire of conventional planning systems. Thus, although it is obviously desirable that an embedded system be capable of forming plans from first principles, it is also important that the system possess a wealth of precompiled procedural knowledge about how to function in the world [6].

1.2.4 Robotic Controllers

A number of systems developed for the control of robots and other real-time processes do have a high degree of reactivity [3, 9]. Such architectures could lead to more viable and robust systems than the traditionally structured planning systems. Yet most of this work has not addressed the issues of general problem-solving and commonsense reasoning; the work is instead almost exclusively devoted to problems of navigation and execution of low-level actions. It remains to extend or integrate these techniques with systems that have the ability to completely change goal priorities, to modify, defer, or abandon current plans, and to reason about what is best to do in light of the current situation.

1.3 Approach

To achieve the kind of behavior discussed above, the architecture of the reasoning system should be both *goal-directed* and *reactive*. That is, while seeking to attain specific goals, the system should also be able to react appropriately to new situations in real time. In particular, it should be able to completely alter focus and goal priorities as circumstances change. In addition, the system should be able to *reflect* on its own reasoning processes. It should be able to choose when to change goals, when to plan and when to act, and how to use effectively its deductive capabilities.

To facilitate cooperation with other systems as well as astronauts and other users, the very design of the system should be based on principles of rational interaction. In particular, it should be possible to ascribe *beliefs*, *goals*, and *intentions* to the system, and to interact with it in terms of these psychological attitudes. In turn, the system itself should be able to reason about and utilize information regarding the beliefs, goals, and intentions of other systems or agents.

SRI International has been conducting research over a number of years into the design of systems having these properties. The set of techniques and concepts are called *Procedural Reasoning Systems* [5, 6, 7, 8].

1.4 This Report

In previous reports to NASA we focussed primarily on the representation and use of procedural knowledge. In this report, we shall concentrate on those aspects of the problem that are critical in the design of *embedded* reasoning and planning systems.

Chapter 2 contains an overview of the Procedural Reasoning System developed at SRI. Chapter 3 describes some critical system functionalities that are required in this and similar applications. Chapter 4 describes the application domain and provides various examples of system operation. This chapter is perhaps the most interesting for those who are not interested in technical details and can be read prior to Chapters 2 and 3. Conclusions are presented in Chapter 5.

Chapter 2

Overview of the Procedural Reasoning System

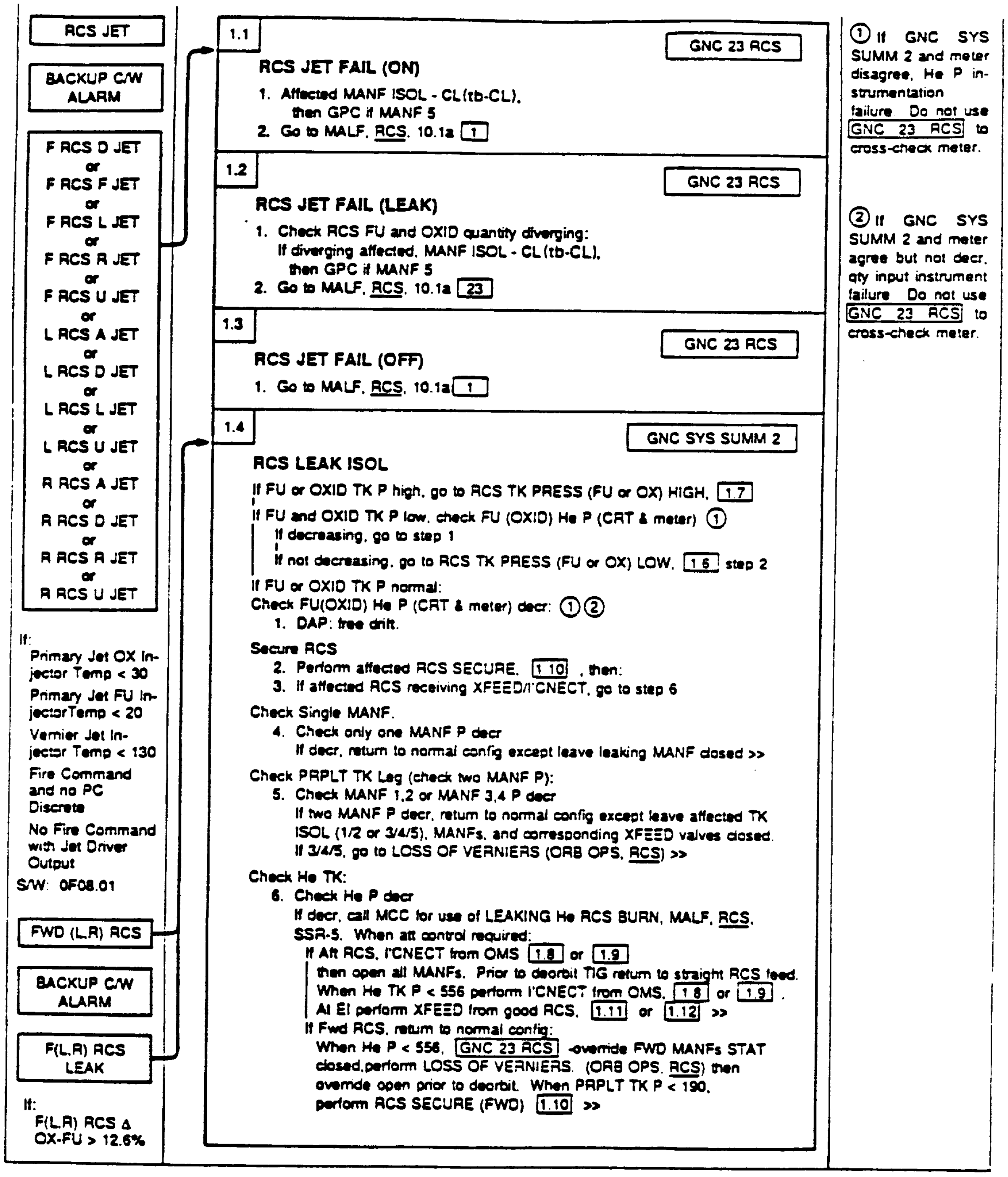
As the sample problem domain, we chose the task of malfunction handling for the Reaction Control System (RCS) of NASA's space shuttle. The shuttle contains three such systems - one forward and two aft. Each is a relatively complex propulsion system that is used to control the attitude of the shuttle. A part of one of the malfunction procedures from NASA's malfunction handling manuals is shown in Figure 2.1. These procedures can be viewed as unelaborated plans of action, and are designed to be executed in a complex and changing environment.

The reasoning and planning system that we applied to this problem is called a *Procedural Reasoning System* (PRS) [6, 8]. PRS consists of a *database* containing current *beliefs* or facts about the world; a set of current *goals* to be realized; a set of *plans* (which, for historical reasons, are called Knowledge Areas or KAs) describing how certain sequences of actions and tests may be performed to achieve given goals or to react to particular situations; and an *intention structure* containing all currently active (executing) KAs. An *interpreter* (or *inference mechanism*) manipulates these components, selecting appropriate plans based on the system's beliefs and goals, placing those selected on the intention structure, and executing them.

The basic structure of PRS is shown in Figure 2.2. The system interacts with its environment (including other systems) through its database (which acquires new beliefs in response to changes in the environment) and through the actions that it performs as it executes its intentions.

2.1 The System Database

The contents of the PRS database may be viewed as representing the current beliefs of the system. Some of these beliefs are provided initially by the system user. Typically, these



RCS JET

BACKUP C/W ALARM

F RCS D JET or F RCS F JET or F RCS L JET or F RCS R JET or F RCS U JET or L RCS A JET or L RCS D JET or L RCS L JET or L RCS U JET or R RCS A JET or R RCS D JET or R RCS R JET or R RCS U JET

If:
Primary Jet OX Injector Temp < 30
Primary Jet FU InjectorTemp < 20
Vernier Jet Injector Temp < 130
Fire Command and no PC Discrete
No Fire Command with Jet Driver Output
S/W: 0F08.01

FWD (L,R) RCS

BACKUP C/W ALARM

F(L,R) RCS LEAK

If:
F(L,R) RCS Δ OX-FU > 12.6%

1.1 GNC 23 RCS

RCS JET FAIL (ON)

1. Affected MANF ISOL - CL(tb-CL), then GPC if MANF 5
2. Go to MALF, RCS, 10.1a 1

1.2 GNC 23 RCS

RCS JET FAIL (LEAK)

1. Check RCS FU and OXID quantity diverging:
   If diverging affected, MANF ISOL - CL(tb-CL), then GPC if MANF 5
2. Go to MALF, RCS, 10.1a 23

1.3 GNC 23 RCS

RCS JET FAIL (OFF)

1. Go to MALF, RCS, 10.1a 1

1.4 GNC SYS SUMM 2

RCS LEAK ISOL

If FU or OXID TK P high, go to RCS TK PRESS (FU or OX) HIGH, 1.7
If FU and OXID TK P low, check FU (OXID) He P (CRT & meter) ①
  If decreasing, go to step 1
  If not decreasing, go to RCS TK PRESS (FU or OX) LOW, 1.6 step 2
If FU or OXID TK P normal:
Check FU(OXID) He P (CRT & meter) decr: ①②
  1. DAP: free drift.
Secure RCS
  2. Perform affected RCS SECURE, 1.10 , then:
  3. If affected RCS receiving XFEED/I'CNECT, go to step 6
Check Single MANF.
  4. Check only one MANF P decr
     If decr, return to normal config except leave leaking MANF closed >>
Check PRPLT TK Leg (check two MANF P):
  5. Check MANF 1,2 or MANF 3,4 P decr
     If two MANF P decr, return to normal config except leave affected TK ISOL (1/2 or 3/4/5), MANFs, and corresponding XFEED valves closed.
     If 3/4/5, go to LOSS OF VERNIERS (ORB OPS, RCS) >>
Check He TK:
  6. Check He P decr
     If decr, call MCC for use of LEAKING He RCS BURN, MALF, RCS, SSR-5. When att control required:
       If Aft RCS, I'CNECT from OMS 1.8 or 1.9
         then open all MANFs. Prior to deorbit TIG return to straight RCS feed.
         When He TK P < 556 perform I'CNECT from OMS, 1.8 or 1.9 .
         At EI perform XFEED from good RCS, 1.11 or 1.12 >>
       If Fwd RCS, return to normal config:
         When He P < 556, GNC 23 RCS -override FWD MANFs STAT closed,perform LOSS OF VERNIERS. (ORB OPS, RCS) then override open prior to deorbit. When PRPLT TK P < 190, perform RCS SECURE (FWD) 1.10 >>

① If GNC SYS SUMM 2 and meter disagree, He P instrumentation failure. Do not use GNC 23 RCS to cross-check meter.

② If GNC SYS SUMM 2 and meter agree but not decr, qty input instrument failure. Do not use GNC 23 RCS to cross-check meter.

Figure 2.1: Portion of an RCS Malfunction Procedure

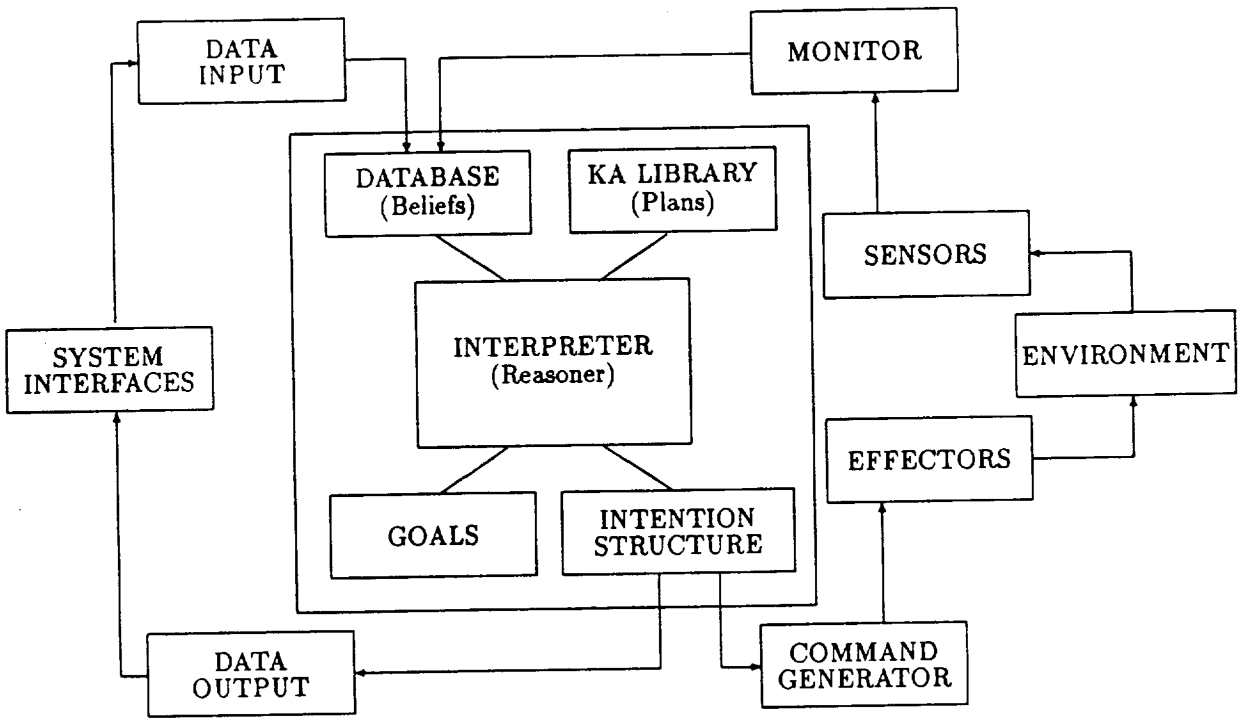


Figure 2.2: Structure of the Procedural Reasoning System

will include facts about static properties of the application domain, such as the structure of some subsystem or the physical laws that must be obeyed by certain mechanical components. Other beliefs are derived by PRS itself as it executes its KAs. These will typically be current observations about the world or conclusions derived by the system from these observations, and these may change over time. For example, at some times PRS may believe that the pressure of an oxidizer tank is within acceptable operating limits, at other times not. Updates to the database therefore necessitate the use of consistency maintenance techniques.

The database itself consists of a set of *state descriptions* describing what is [believed to be] true at the current instant. We use first-order predicate calculus for the state description language. State descriptions can contain variables (implicitly assumed to be universally quantified)[1] and the usual logical connectives ($\wedge$, $\vee$, and $\neg$, representing respectively conjunction, disjunction, and negation). A sample set of database beliefs for the RCS application is given below.

```
(type manifold frcs-ox-manf-1)
(type p-xdcr frcs-ox-manf-1-p-xdcr)
(type valve frcs-fu-tk-isol-12-valve)
```

[1] Logical variables are represented in PRS by names with a $ prefix (see Section 3.12).

```
(part-of frcs-ox frcs-ox-manf-1)
(associated-unit frcs-ox-manf-1-p-xdcr frcs-ox-manf-1)
(connects frcs-ox-tk-isol-12-valve frcs-ox-tk frcs-ox-tk-12-leg)

(status frcs-ox-manf-1-p-xdcr good)
(position frcs-ox-tk-isol-12-valve op)
(value frcs-ox-manf-1-p-xdcr 245)
```

The first three facts represent type information. For example, the third fact states that the object named **frcs-fu-tk-isol-12-valve** is a valve. The next three facts represent structural information, describing in this case a part-whole relationship, the positioning of a sensor, and a valve connection within the system. The last three facts describe dynamic facts that represent the current status and parameter values of the system. In the application explored here, over 650 such facts are utilized for the forward RCS alone.

State descriptions that describe *internal* system states are called *metalevel* expressions. The basic metalevel predicates and functions are predefined by the system. For example, the metalevel expression (*goal g) is true if g is a current goal of the system.[2].

Given such a database, the diagnostic procedures can make use of this information to perform what might be considered simple commonsense tasks for an astronaut. For example, the instruction to "return to normal configuration except leave affected tank isolation valves, manifolds, and corresponding cross-feed valves closed" (Figure 2.1) can be directly represented and reasoned about. In particular, the instruction can be represented in a way that is impervious to system reconfiguration, is not hard-wired to particular identifiers, and can be used for any of the three reaction control systems on the shuttle.

2.2 Goals

Unlike most AI planning and reasoning systems, PRS goals represent desired *behaviors* of the system, rather than static world states that are to be eventually achieved. Hence goals are expressed as conditions over some interval of time (i.e., over some sequence of world states). A given action (or sequence of actions) is said to *succeed* in achieving a given goal if its execution results in a behavior that satisfies the goal description.

Goal behaviors can be described by applying a temporal operator to a state description. Three temporal operators are currently being used. The expression (!*p*), where *p* is some

[2]We adopt the convention that metalevel predicates that are prefixed with an asterisk denote attitudes (e.g., goals, beliefs, or intentions) of the system itself; attitudes of other agents are represented by predicates without the asterisk and take an additional argument (the name of the agent). For historical reasons, we also use the predicate *fact synonymously with *belief

state description (possibly involving logical connectives), is true of a sequence of states if p is true of the last state in the sequence; that is, it denotes those behaviors that *achieve* p. Thus we might use the behavior description `(!(position frcs-fu-tk-isol-12-valve cl))` to describe the goal to close valve `frcs-fu-tk-isol-12-valve`.

The expression (?p) is true of a sequence of states if p is true of the first state in the sequence. That is, it can be considered to denote those behaviors that result from a successful *test* for p. Let's examine this notion more carefully. We call an action a test for a condition p if successful completion of the action guarantees that p were true just prior to beginning the action. For example, imagine an action a such that, if p were true just prior to executing a, a certain condition, q say, would be observed at its completion (e.g., a might be the action of dipping litmus paper in a solution, q the condition that the paper be red, and p the condition that the solution be acidic). Clearly, if q is observed, the sequence of states involved in performing a must have been such that p is true of the first of these states. On the other hand, if q is not observed, p may not be true of the first state in the sequence. Thus, a will count as a test for p if it signals success when q is observed and signals failure otherwise.

Finally, (#p) if true if p is preserved (is maintained invariant) throughout the sequence. Usually, when one establishes a goal of maintenance, it is intended that that goal be maintained until some condition becomes true. Thus, goals of maintenance usually appear in the form (#p $\wedge$!q), meaning to preserve p until q is achieved.

Behavior descriptions can be combined by means of the logical operators $\wedge$ and $\vee$, representing, respectively, the conjunction and disjunction of the component expressions. Existentially quantified variables are represented by symbols prefixed with a $ sign.

As with state descriptions, behavior descriptions are not restricted to describing the external environment, but can also characterize the internal behavior of the system. Such behavior specifications are called metalevel behavior specifications. One important metalevel behavior is described by an expression of the form (=> p). This specifies a behavior that places the state description p in the system database. Another way of describing this behavior is (!(*belief p)).

2.3 Knowledge Areas

Knowledge about how to accomplish given goals or react to certain situations is represented in PRS by declarative procedure specifications called *Knowledge Areas* (KAs). Each KA consists of a *body*, which describes the steps of the procedure, and an *invocation condition*, which specifies under what situations the KA is useful. Together, the invocation condition and body of a KA express a declarative fact about the results and utility of performing certain sequences of actions under certain conditions [6].

The body of a KA is represented as a graphic network and can be viewed as a plan or plan schema. Each arc of the network is labeled with a goal to be achieved by the system. This is in contrast to traditional programming languages or flowcharts, which label each step either with some primitive command or some specific subroutine to be called. By establishing goals rather than calling named subroutines, the system is able to reason about and choose the most effective means for accomplishing those goals *in the given circumstances.* This allows PRS to react appropriately in a great range of situations and in the presence of degraded or uncertain information. Moreover, the knowledge expressed in any given KA is largely independent of other KAs, thereby providing a very high degree of modularity and verifiability. It is thus possible to build domain knowledge incrementally, with each component KA having a well-defined and easily understood semantics.

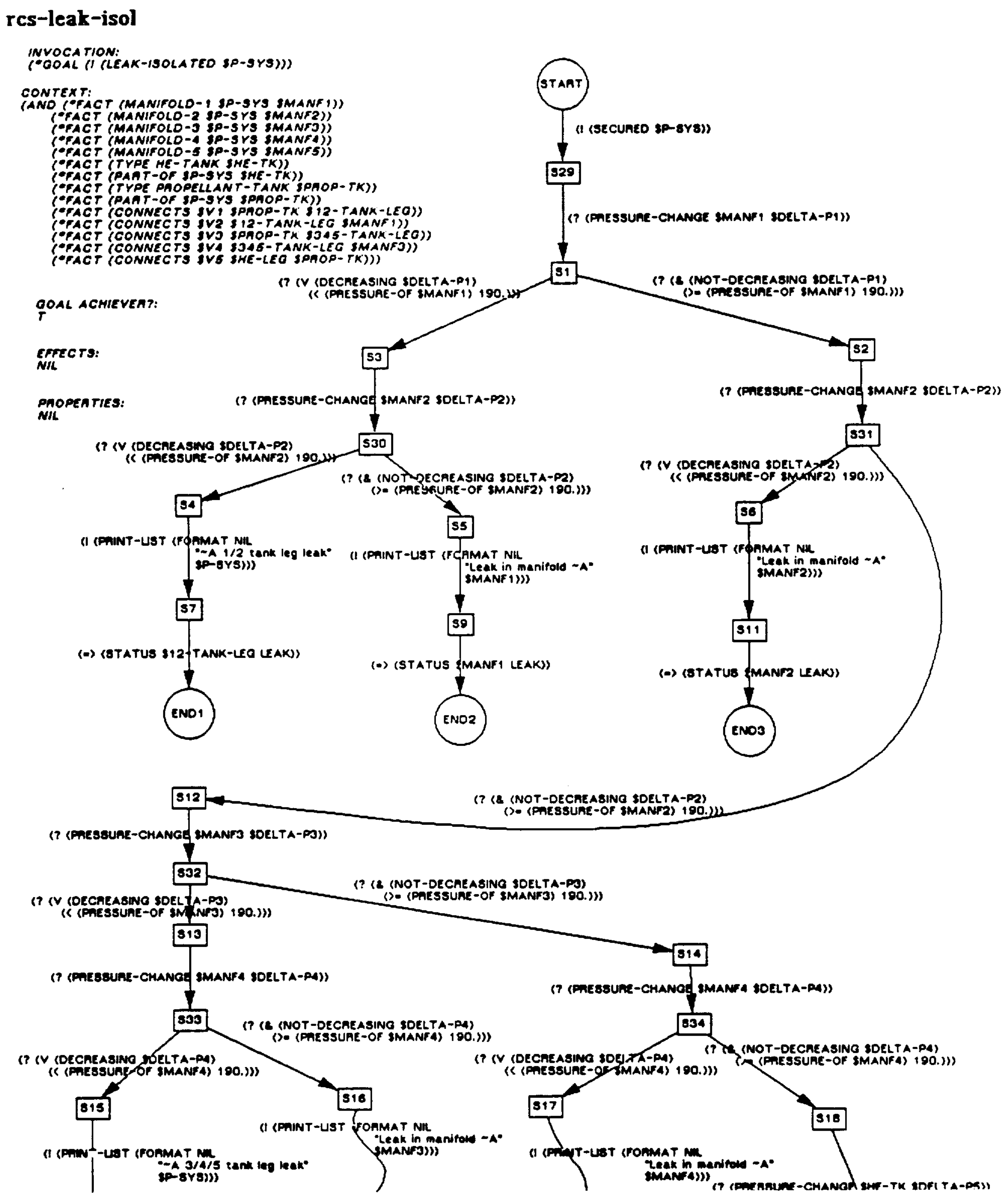
The invocation condition has two components: a *triggering part* and a *context part.* Both must be satisfied for the KA to be invoked. The triggering part is a logical expression describing the *events* that must occur for the KA to be executed. Usually, these consist of some *change* in system goals (in which case, the KA is invoked in a goal-directed fashion) or system beliefs (resulting in data-directed or reactive invocation), and may involve both. The context part is a logical expression specifying those conditions that must be true of the current state for the KA to be executed.

There are some properties of KAs that are crucial for the correct functioning of the system. For example, if the KA is invoked by the establishment of some new goal, it is important to know whether or not successful execution of the KA (i.e., reaching an end-node) realizes that goal. These KA properties could be placed in the system database along with the other facts that are pertinent to the application domain. However, for convenience we place these facts in predefined *slots* in the KA structure itself.

Similarly, KA properties that are not essential to the functioning of the interpreter but which may be required by application-specific metalevel KAs can either be placed in the system database or in a special *property slot* of each KA. Such properties, for example, might include information on the likelihood of success of the KA or its average execution time.

A typical example of part of a KA is given in figure 2.3. It describes a procedure to isolate a leak in an RCS. The invocation part describes under what conditions this KA is useful. In this case, the KA is considered useful whenever the system acquires the goal to isolate a leak in an RCS (`$p-sys`), provided the various type and structural facts given in the context part are true. (In determining the truth value of the invocation part, some of the variables appearing in the invocation part will be bound to specific identifiers. Indeed, in this case, all the variables will be so bound.)

The KA body describes what to do if the KA is chosen for execution. Execution begins at the `start` node in the network, and proceeds by following arcs through the network. Execution completes if execution reaches a finish node (a node with no exiting arcs). If


rcs-leak-isol

```
INVOCATION:
(*GOAL (! (LEAK-ISOLATED $P-SYS)))

CONTEXT:
(AND (*FACT (MANIFOLD-1 $P-SYS $MANF1))
     (*FACT (MANIFOLD-2 $P-SYS $MANF2))
     (*FACT (MANIFOLD-3 $P-SYS $MANF3))
     (*FACT (MANIFOLD-4 $P-SYS $MANF4))
     (*FACT (MANIFOLD-5 $P-SYS $MANF5))
     (*FACT (TYPE HE-TANK $HE-TK))
     (*FACT (PART-OF $P-SYS $HE-TK))
     (*FACT (TYPE PROPELLANT-TANK $PROP-TK))
     (*FACT (PART-OF $P-SYS $PROP-TK))
     (*FACT (CONNECTS $V1 $PROP-TK $12-TANK-LEG))
     (*FACT (CONNECTS $V2 $12-TANK-LEG $MANF1))
     (*FACT (CONNECTS $V3 $PROP-TK $345-TANK-LEG))
     (*FACT (CONNECTS $V4 $345-TANK-LEG $MANF3))
     (*FACT (CONNECTS $V5 $HE-LEG $PROP-TK)))

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL
```

START
(! (SECURED $P-SYS))
S29
(? (PRESSURE-CHANGE $MANF1 $DELTA-P1))
S1
(? (V (DECREASING $DELTA-P1) (< (PRESSURE-OF $MANF1) 190.)))
(? (& (NOT-DECREASING $DELTA-P1) (>= (PRESSURE-OF $MANF1) 190.)))
S3
S2
(? (PRESSURE-CHANGE $MANF2 $DELTA-P2))
(? (PRESSURE-CHANGE $MANF2 $DELTA-P2))
S30
S31
(? (V (DECREASING $DELTA-P2) (< (PRESSURE-OF $MANF2) 190.)))
(? (& (NOT-DECREASING $DELTA-P2) (>= (PRESSURE-OF $MANF2) 190.)))
(? (V (DECREASING $DELTA-P2) (< (PRESSURE-OF $MANF2) 190.)))
S4
S5
S6
(! (PRINT-LIST (FORMAT NIL "~A 1/2 tank leg leak" $P-SYS)))
(! (PRINT-LIST (FORMAT NIL "Leak in manifold ~A" $MANF1)))
(! (PRINT-LIST (FORMAT NIL "Leak in manifold ~A" $MANF2)))
S7
S9
S11
(=> (STATUS $12-TANK-LEG LEAK))
(=> (STATUS $MANF1 LEAK))
(=> (STATUS $MANF2 LEAK))
END1
END2
END3
S12
(? (& (NOT-DECREASING $DELTA-P2) (>= (PRESSURE-OF $MANF2) 190.)))
(? (PRESSURE-CHANGE $MANF3 $DELTA-P3))
S32
(? (& (NOT-DECREASING $DELTA-P3) (>= (PRESSURE-OF $MANF3) 190.)))
(? (V (DECREASING $DELTA-P3) (< (PRESSURE-OF $MANF3) 190.)))
S13
S14
(? (PRESSURE-CHANGE $MANF4 $DELTA-P4))
(? (PRESSURE-CHANGE $MANF4 $DELTA-P4))
S33
S34
(? (& (NOT-DECREASING $DELTA-P4) (>= (PRESSURE-OF $MANF4) 190.)))
(? (V (DECREASING $DELTA-P4) (< (PRESSURE-OF $MANF4) 190.)))
(? (V (DECREASING $DELTA-P4) (< (PRESSURE-OF $MANF4) 190.)))
(? (& (NOT-DECREASING $DELTA-P4) (>= (PRESSURE-OF $MANF4) 190.)))
S15
S16
S17
S18
(! (PRINT-LIST (FORMAT NIL "Leak in manifold ~A" $MANF3)))
(! (PRINT-LIST (FORMAT NIL "~A 3/4/5 tank leg leak" $P-SYS)))
(! (PRINT-LIST (FORMAT NIL "Leak in manifold ~A" $MANF4)))
(? (PRESSURE-CHANGE $HE-TK $DELTA-P5))

Figure 2.3: Portion of a KA for Leak Isolation

more than one arc emanates from a given node, any one of the arcs emanating from that node may be traversed. To traverse an arc, the system must either (1) determine from the database that the goal has already been achieved or (2) find a KA (procedure) that achieves the goal labelling that arc. For example, to traverse the arc emanating from the start node requires either that the system be already secured or that some KA for securing the RCS be found and successfully executed. Similarly, to transit the next arc requires that some KA be found for determining the pressure change (`$delta-p1`) in the manifold `$manf1`. If the system fails to traverse an arc emanating from some node, other arcs emanating from that node may be tried. If, however, the system fails to achieve any of the goals on arcs emanating from the node, the KA as a whole will fail. For example, since only one arc emanates from the start node in Figure 2.3, if all attempts to secure the RCS fail, this procedure for isolating a leak in the system will also fail. The full KA for this procedure consists of over 45 nodes and is the largest in the system.

Important properties of the KA are represented in the slots on the left side of the KA structure. For example, the `goal achiever` slot is set to T (true), representing the fact that, upon successfully completing this KA, the goal that triggered execution will have been achieved.

Some KAs have no bodies. These are the primitive KAs of the system and have associated with them some primitive action that is directly performable by the system. Clearly, execution of any KA must eventually reduce to the execution of sequences of primitive KAs (unless, of course, each of the subgoals of the KA has already been achieved).

The set of KAs in a PRS application system not only consists of procedural knowledge about a specific domain, but also includes *metalevel* KAs — that is, information about the manipulation of the beliefs, desires, and intentions of PRS itself. For example, typical metalevel KAs encode various methods for choosing among multiple applicable KAs, determining how to achieve a conjunction or disjunction of goals, and computing the amount of additional reasoning that can be undertaken, given the real-time constraints of the problem domain. In achieving this, these metalevel KAs make use of information about KAs that is contained in the system database or in the property slots of the KA structures.

In the application described herein, about 70 object-level KAs were used together with about 15 metalevel KAs.

2.4 The Intention Structure

The intention structure contains all those tasks that the system has adopted (chosen) for execution, either immediately or at some later time. These adopted tasks are called *intentions*. A single intention consists of some top-level KA or goal, together with all the various [sub-] KAs that are currently being used as means to fulfilling the requirements of the top-level KA. However, at any given moment, the intention structure may contain a number of such intentions, some of which may be suspended or deferred, some of which may be waiting for certain conditions to hold prior to activation, and some of which may be metalevel intentions for deciding among various alternative courses of action.

For example, in handling a malfunction in the RCS the system might have, at some instant, three tasks (intentions) on the intention structure: one suspended while waiting for, say, the fuel-tank pressure to decrease below some designated threshold; another suspended after having just posted some goal that is to be accomplished (such as the securing of the RCS); and the third (a metalevel procedure) being executed to decide which way to accomplish that goal.

The order in which these intentions are executed is determined by metalevel KAs which themselves must be adopted as intentions to become effective. This metalevel control allows reasoning in arbitrarily complex ways about the scheduling of these tasks, while retaining the ability to respond quickly and appropriately to new goals and beliefs.

2.5 The System Interpreter

The PRS interpreter runs the entire system. From a conceptual standpoint, it operates in a relatively simple way. At any particular time, certain goals are active in the system and certain beliefs are held in the system database. Given these extant goals and beliefs, a subset of KAs in the system will be applicable (i.e., relevant). One or more of these applicable KAs will then be chosen for execution by placing them on the intention structure.

In determining KA applicability, the interpreter will not automatically perform any deduction. Both beliefs and goals are matched by using unification only. This allows appropriate KAs to be selected very quickly and guarantees a certain degree of reactivity. If we allowed arbitrary deductions to be made, we could no longer furnish such a guarantee. However, PRS is always able to perform any deductions it chooses by invoking appropriate metalevel KAs. These metalevel KAs are themselves interruptible, so that the reactivity of the system is retained.

In the course of executing the chosen KA, new subgoals will be posted and new beliefs derived. Changes in the environment may also modify the existing beliefs of the system. When new goals are established, the interpreter checks to see if any new KAs are relevant, chooses one or more, places them on the intention structure, selects an item from the intention structure, and begins executing it. Likewise, whenever a new belief is added to the database, the interpreter will perform appropriate consistency maintenance procedures and possibly activate other relevant KAs. During this process, various metalevel KAs may also be called upon to make choices among alternative paths of execution, choose among multiple applicable KAs, decide what intentions to execute next, decompose composite goals into achievable components, and make other decisions.

Unless some new belief or goal activates some new KA, PRS will try to fulfill any intentions it has previously decided upon. But if some important new fact or goal does become known, PRS will reassess its current intentions, and perhaps choose to work on something else. Thus, not all options that are considered by PRS arise as a result of means-end reasoning. Changes in the environment may lead to changes in the system's beliefs, which in turn may result in the consideration of new plans that are not means to any already intended end. PRS is therefore able to *change its focus completely* and pursue new goals when the situation warrants it. In many space operations, this happens quite frequently as emergencies of various degrees of severity occur in the process of handling other, less critical tasks. PRS can even alter its intentions regarding its own reasoning processes – for example, it may decide that, given the current situation, it has no time for further reasoning and so must act immediately.

2.6 Multiple Asynchronous Systems

In some applications, it is necessary to monitor and process many sources of information at the same time. Because of this, PRS was designed to allow several instantiations of the basic system to run in parallel. Each PRS instantiation has its own database, goals, and KA library, and operates asynchronously relative to other PRS instantiations. Communication among the various PRS instantiations is achieved by message passing. The messages are written into the database of the receiving PRS, which must then decide what to do, if anything, with the new information. As a rule, this decision is made by a fact-invoked KA (in the receiving PRS), which responds upon receipt of the external message. In accordance with such factors as the reliability of the sender, the type of message, and its own beliefs, goals, and current intentions, the receiver determines what to do about the message — for example, to acquire a new belief, establish a new goal, or modify an existing intention.

In this particular application, two instances of PRS were set up. One, called INTERFACE, handles most of the low-level sensor readings, controls effectors, and checks for faults in these components. The other, called misleadingly RCS, contains most of the high-level malfunction procedures, much as they appear in the malfunction handling manuals for the shuttle.

As an example of the communication between these two systems, consider the case in which INTERFACE wishes to advise RCS that the valve `frcs-fu-tk-isol-12-valve` is closed. To do so, it would send RCS the message

```
(asserted INTERFACE (position frcs-fu-tk-isol-12-valve cl))
```

RCS could then choose what to do with this message, given appropriate KAs for responding to it. Note that the belief that the valve is closed is not directly inserted into the database of the recipient. In complex domains in which processes or agents may be unreliable, it is preferable to store the fact that some agent (INTERFACE in this case) has *asserted* something, without committing to believing that assertion. The recipient then has the opportunity (using appropriate KAs) to accept the asserted fact and add it to its database, to reject it as unreliable, or to combine it with other evidence in some other way.

For similar reasons, when some PRS agent, *A*, wants another PRS agent, *B*, to adopt some particular goal, the only way this can be effected is by passing *B* a message that requests that *B* establish the given goal. That is, *A* cannot directly establish a goal for *B*; the best *A* can do is to get *B* to believe that *A* desires that *B* adopt the given goal. For example, if `RCS` wished the `INTERFACE` to close a valve `frcs-fu-tk-isol-12-valve`, `RCS` would send `INTERFACE` the message

```
(requested RCS (!(position frcs-fu-tk-isol-12-valve cl)))
```

Chapter 3

Critical Features of the System

In this chapter, we review in greater depth some of the features of PRS that are critical to its successful operation as an embedded real-time reasoning system.

3.1 Invocation of KAs

The applicability of a KA is specified by means of its associated invocation condition. The invocation condition consists of two parts:

1. A logical expression specifying some pattern of *initiating events*
2. A logical expression specifying the *context* of invocation.

An initiating event is the acquisition of a new belief or the establishment of a new goal or intention. For example, if the initiating condition of a given KA were
`(*fact (position frcs-fu-tk-isol-12-valve cl))`,
the KA would be invoked whenever the system *acquired* the belief that the `frcs-fu-tk-isol-12-valve` was closed. Thus, it is the *change* in the system's beliefs that triggers the KA. Similarly, if the initiating condition were
`(*goal (!(secured frcs)))`,
the KA would be invoked upon the system *acquiring* that goal.

The context specifies additional conditions that must be true for the KA to be invoked, once it has been triggered by some initiating event. Thus, if the context of a KA were `(*fact (position frcs-fu-he-tk-A-valve cl))`, the KA could only be invoked if the system believed that the `frcs-fu-he-tk-A-valve` were closed.

Let's say that I is the initiating condition for a given KA and C is its invocation context. Furthermore, let us assume that at moment t the system's state (i.e., its beliefs, goals, and intentions) is S and at the next moment t' the state is S'. Then the KA will be invoked if and only if

$$(S' - S) \vdash I \text{ and } S' \vdash C$$

where $x \vdash y$ is true if the truth of expression y can be directly deduced upon unifying each of y's components with the components of x.

It is important to note that the only way a KA can be invoked is by the occurrence of some initiating event or change in the system's beliefs, goals, or intentions. Should such an event trigger a KA, and that KA not be immediately adopted as an intention, then that event cannot re-trigger the KA on subsequent cycles. (Of course, some subsequent occurrence of the same type of event could trigger the KA afresh.) What are the consequences of this?

Imagine that two different alarms, alarm-1 and alarm-2, are sounded at once, each triggering a different KA (say, KA-1 and KA-2, respectively). As the system has more than one applicable KA from which to choose, one or more meta-KAs will be invoked to determine what to do. If, as a result of the metalevel processing, one of these KAs is chosen for adoption (say KA-1), the other invoked KA will simply be discarded. In this case, although the system would "know" that alarm-2 had sounded (it would be in the system database), it would take no action with respect to that alarm. This need not mean that it would never take any action in response to that alarm, although usually this would be so. For example, there may be some KA that is invoked every now and then to check on things that have been left unattended. Such a KA could notice that alarm-2 was on, that nothing had been done about it, and then, indirectly, invoke KA-2 to respond to it.

The other possibility, dependent on the metalevel processing, is that *both* KAs are adopted as intentions. In this case, both alarms will be attended to. The order in which KA-1 and KA-2 are evaluated will depend on the ordering of their corresponding intentions on the intention structure; for example, both could be pursued in parallel, or one could be deferred until the other was finished. We say more about the intention structure below.

In summary, the system must respond to events — and form intentions appropriate to those events — *as they occur*. As changes to elements of the system's state are, in most cases, rare in comparison to the total number of state elements, system efficiency (and response time) is thereby substantially enhanced. On the other hand, as at any time the current state of the system is determined entirely by its history of changes,[1] we loose no deductive capabilities by this more restricted form of KA activation.

[1]This assumes an empty initial state. But if the initial state, i say, is not empty, we can transform the problem into an equivalent one that has an empty initial state and that begins with an event that directly brings about i.

3.2 Goals and Intentions

Goals are of two kinds: *intrinsic* goals and *operational* goals. Intrinsic goals[2] are those that the system acquires directly from some specific sources. The only source of intrinsic goals for the shuttle implementation is the user (astronaut or mission controller), who has the power to impose arbitrary goals on the system. In other implementations, one could envisage other sources capable of imposing such goals. For example, an autonomous system being controlled by PRS might allow special sensors to generate intrinsic goals, such as to recharge batteries when battery power decreases below a certain threshold, or to escape in the presence of an overwhelming foe.

Operational goals are those that the system acquires in attempting to fulfill some intention. That is, operational goals are those that are established during execution of a KA that has been previously adopted as an intention. Thus, operational goals are always means to some end, although that end may not always be explicit. For example, when a KA is invoked by the occurrence of a new system goal, g, all goals in the body of the KA will be means toward achieving g. On the other hand, when a KA is invoked by the occurrence of a new system belief, the goals appearing in the KA are means of responding to the aquisition of that belief, but the end result the system is aiming to achieve is left unspecified. For example, upon the activation of some alarm, a KA that diagnoses the fault and corrects it might be invoked. The reason for invoking this KA (presumably, to maintain the integrity of the space-craft) is not specified anywhere, yet each of the goals occurring in this KA are means to that (unspecified) end.

A KA invoked by the acquisition of some intrinsic goal or by some change in system beliefs can give rise, if adopted by the system, to a *new* system intention. In this case, the initiating event is called the *purpose* of the intention and the KA so invoked is called the *head* of the intention. As the head KA is executed, it will give rise to various operational goals. The KAs that respond to these operational goals will form part of the originating intention, together with any KAs that these KAs in turn invoke, and so on. Thus, a single intention consists of a head KA (invoked either by an intrinsic goal or new belief), together with the various other KAs that are utilized in attempting to execute the head KA.

3.3 The Establishment and Removal of Goals

Intrinsic goals are established by specific external sources and, as with beliefs, must be responded to immediately if at all. As with beliefs, they will be remembered, even if no intention is formed to accomplish them. They will be removed if explicitly requested (by the external source that established them) or if the system comes to believe that they are accomplished (either through its own efforts or those of some other agent).

[2]Some philosphers would call these *desires*.

Operational goals are established by the attempted execution of some KA that is part of some intention of the system. Should the system attempt to achieve an operational goal and *fail*, that goal will be reestablished, and another attempt made to achieve it. This will continue until the system comes to believe either that the goal is accomplished, through its own efforts or those of some other agent, or that the goal cannot be readily accomplished. Once this state is reached, the goal will be dropped.

This raises two related issues: what other attempts are made to achieve the goal and how does the system come to believe a goal cannot be readily accomplished? We shall answer the latter question first. There are two ways the system can come to believe that a goal cannot be accomplished. One way is to deduce it, but this will depend on the provision of appropriate metalevel KAs for performing the deduction and upon their being invoked. The other way is simply to fail in all attempts at achieving it.

This brings us to the former question — what attempts does the system make to achieve a given goal? The system currently tries, exactly once, every possible KA instance that can possibly achieve the goal. It does not ask that previously achieved goals be reachieved (in some other way), nor does it try the same KA instance more than once. In this sense, it is equivalent to a "fast-backtrack" parser [13]. There is good reason for at least the first of these choices.

Unlike planning some course of action or parsing sentences, once some goal has been achieved there is no point in trying other ways to achieve it, even if these may benefit attempts to achieve subsequent goals. Or more accurately, there is no reason to *constrain* oneself to look only at other ways to achieve this goal. For example, consider that we wish to achieve some goal f and then some goal g. Let's say that, in achieving f, some condition p is made true (perhaps as a side effect of the actions taken to achieve f). Unfortunately, with p true, it proves impossible to achieve g with the KAs at our disposal (because, let's imagine, $\neg p$ is part of the context of the invocation part of all KAs that achieve g). Now, a planner or full-backtrack parser might reattempt to achieve f some other way, in the hope that it would succeed without incurring the troublesome side-effect. But PRS has *actually* achieved f - and also p - so that there is no use trying other methods to reachieve that goal! Instead, one should now try to *achieve* $\neg p$, preferably while *maintaining* f.

Now, either there is a way to achieve $\neg p$ or there is not. If there is not, there is nothing we can do. If we can achieve $\neg p$, and we are content to try it, then we could rewrite the KAs that achieve g to have $\neg p$ as their first subgoal, rather than as a constraint on invocation. In this way, it is possible to write all KAs in a form whereby forbidding PRS to retry successfully achieved goals does not restrict its capabilities.

The remaining issue concerns the number of tries we make of a single KA instance to achieve a given goal. It is, of course, quite possible that, where the first try does not succeed, the next will, even if we carry out exactly the same actions as we did the first time. However, to determine if retrying could succeed would, in most practical cases, require knowledge of the state of the world that goes well beyond that available to the system. Assuming the lack of such knowledge, we took the reasonable course of allowing at most one try for each KA instance. This clearly affects the capabilities of the system – in some cases, it may be that trying twice would succeed where trying once does not.

Once all attempts at achieving a given goal have in this way been exhausted, it is still possible for some metalevel KA to respond to this failure and invoke yet other means to achieve the goal. For example, a meta-KA could invoke certain deductive machinery, or could decide to retry some KA instances that, although having been tried once, appear (for some reason known to the meta-KA) to be worth trying again.

3.4 Goals over Sets of Objects

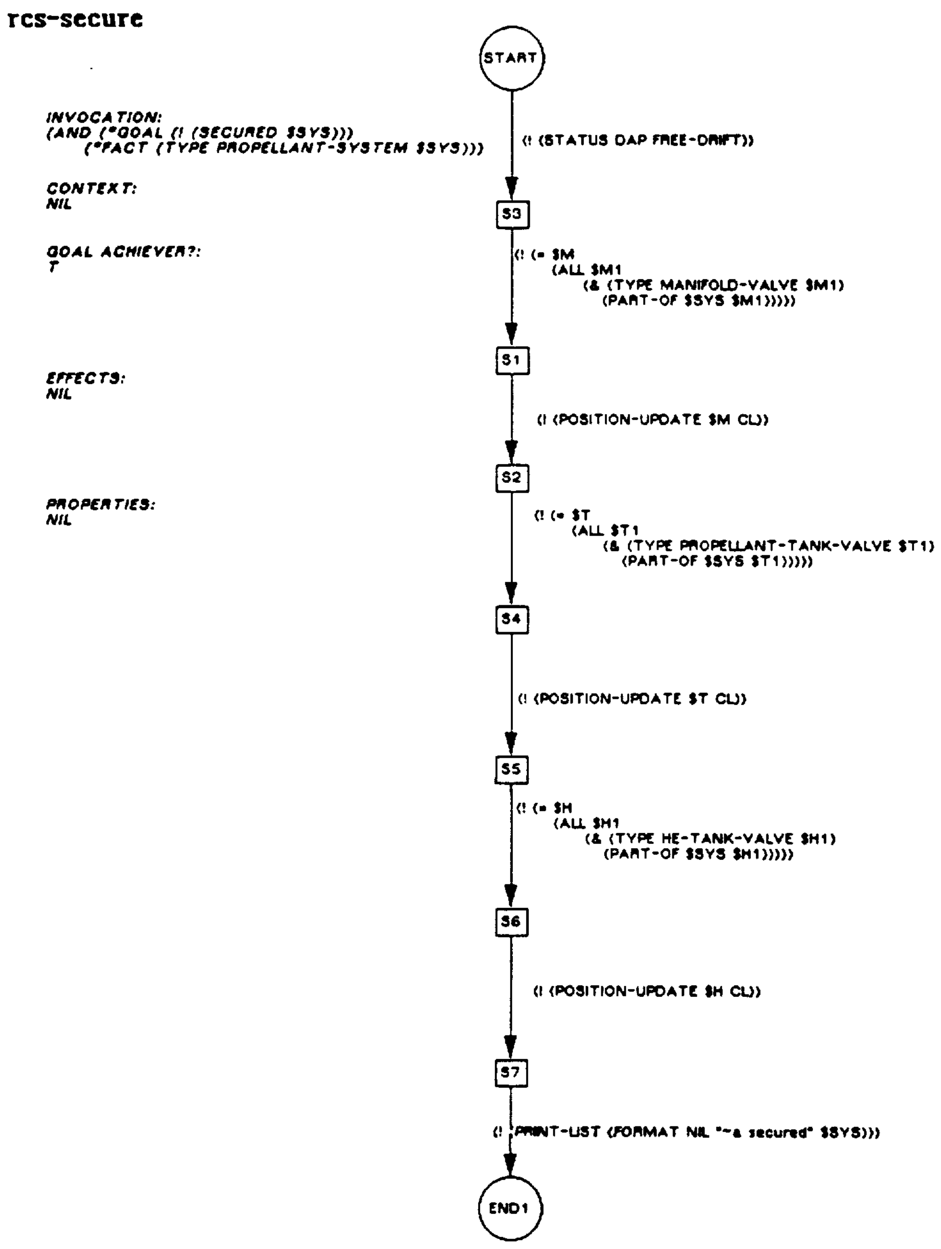
As so far described, goals can be one of three types, denoted `(!`p`)`, `(?`p`)`, and `(#`p`)`, where p is some condition of the state of the world. This condition has been restricted to formulas of first-order predicate calculus, possibly containing existentially-quantified variables where the scope of the existential includes the entire expression.

For example, consider the arc label `(!(P $x $y))` where **$x** has been bound to `(f A $z)` and **$y** and **$z** are unbound. The meaning of this expression is to try to achieve a state that satisfies ∃ **$y $z** . `(P (f A $z) $y)`. This is quite sufficient for most tasks, such as the closing of particular valves or the determination of particular transducer readings.

However, there are many other tasks for which it is convenient to quantify over *all* objects having a certain property. For example, one might want to turn off all the lights in the house, or close all affected manifolds. It would be convenient to represent such conditions by expressions of the form `({ $y | p($y)} g($y))`, where `p($y)` is a formula containing at least one free occurrence of **$y** and **g($y)** is a goal formula also containing a free occurrence of **$y**. The meaning of this expression is that, for all **$y** that satisfy `p($y)`, the goal **g($y)** is to be achieved.

As currently implemented, PRS requires two steps to achieve this effect. The first collects the set of objects upon which the action is to take effect, and the second then applies the action to elements of that set. Moreover, for each action that we wish to operate on a set, we must specify a new KA that is specifically designed for that purpose. An example of this approach is shown in Figure 3.4. In this case, it is necessary to close all the valves of the system in a specified order: first the manifold valves, then the propellant-tank valves, and finally the helium-tank valves. This is clearly a cumbersome way to represent such operations. We expect that subsequent implementations of PRS will use a form more similar to that described in the paragraph above.

One has to be careful with the set operator. First, the set of objects must satisfy the specified condition at the beginning of the interval over which the composite action is



```
INVOCATION:
(AND (*GOAL (! (SECURED $SYS)))
     (*FACT (TYPE PROPELLANT-SYSTEM $SYS)))

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL
```

```
(! (STATUS DAP FREE-DRIFT))

(! (= $M
      (ALL $M1
           (& (TYPE MANIFOLD-VALVE $M1)
              (PART-OF $SYS $M1)))))

(! (POSITION-UPDATE $M CL))

(! (= $T
      (ALL $T1
           (& (TYPE PROPELLANT-TANK-VALVE $T1)
              (PART-OF $SYS $T1)))))

(! (POSITION-UPDATE $T CL))

(! (= $H
      (ALL $H1
           (& (TYPE HE-TANK-VALVE $H1)
              (PART-OF $SYS $H1)))))

(! (POSITION-UPDATE $H CL))

(! (PRINT-LIST (FORMAT NIL "~a secured" $SYS)))
```

Figure 3.1: A KA with Goals over Sets of Objects

performed. Second, the system will not assume that it knows about *all* objects that can possibly satisfy a given predicate unless the predicate is specified to be *closed-world.* In the case that the predicate is not so specified, a KA must be found that matches the composite goal directly. (It is not necessary that such KAs be able to identify explicitly all the specified objects. For example, one can turn off all the lights in a house without identifying them simply by turning off the power supply to the house).

Third, any existentially quantified variable appearing in p($y) will not retain its binding beyond the scope of the entire goal expression. This is because its binding may depend on the binding of the set variable **$y**, and this information is only retained during the execution of the goal g($y).

3.5 Conditional Intentions

Sometimes it is desirable to form the intention that, when some condition occurs, some action should be performed. We call these *conditional intentions.* Typically, we may get to a certain point in a procedure (KA) where we want to suspend execution until some condition is found to be satisfied.

For example, after having switched from a faulty regulator to a properly functioning one, we may want to switch control of the regulator valve to the on-board computers (GPC). However, this cannot be done until the pressure in the system drops below 312 psi, as otherwise the computers will assume a failure in the new regulator and shut it off. Thus, one would like to be able to suspend further execution of the procedure until the pressure dropped below 312 psi, after which execution can proceed.

In PRS, conditional intentions are created by means of certain metalevel KAs. Their effect is to suspend execution of some given intention, which then remains in a dormant state until the appropriate *activation condition* (in the above case, a pressure below 312 psi) is satisfied (see Section 3.6).

Conditional intentions arise in many other situations; indeed, it would be hard to imagine an effective real-time reasoning system that did not require them. For example, conditional intentions are necessary in the following cases:

- When certain events may be indicative of a problem but allowance has to be made for transient effects; one often has to wait until any transients die away.
- When sampling parameters over time, such as when a *rate* of change is required.
- When waiting for a reply from a request to another system or a user; it is often desirable to suspend the process making the request until an answer is received, and if not received within a reasonable time to adopt some other course of action.

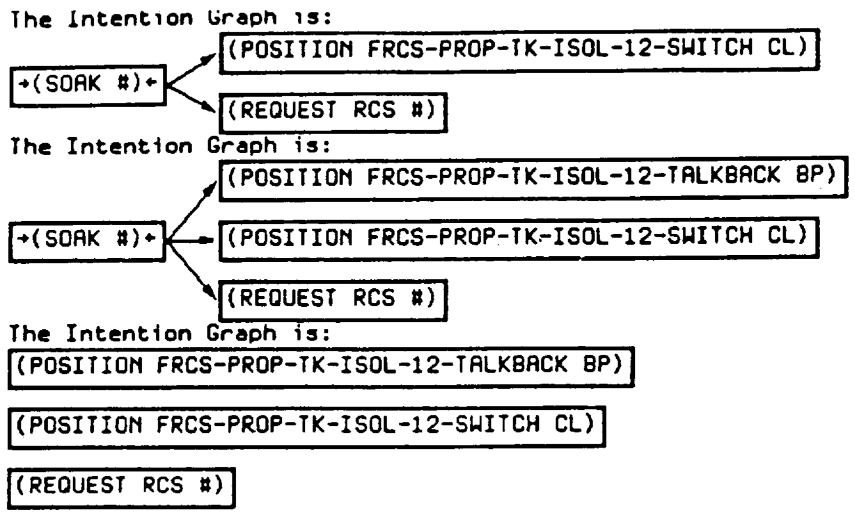

The Intention Graph is:
→(SOAK #)←
(POSITION FRCS-PROP-TK-ISOL-12-SWITCH CL)
(REQUEST RCS #)
The Intention Graph is:
→(SOAK #)←
(POSITION FRCS-PROP-TK-ISOL-12-TALKBACK BP)
(POSITION FRCS-PROP-TK-ISOL-12-SWITCH CL)
(REQUEST RCS #)
The Intention Graph is:
(POSITION FRCS-PROP-TK-ISOL-12-TALKBACK BP)
(POSITION FRCS-PROP-TK-ISOL-12-SWITCH CL)
(REQUEST RCS #)

Figure 3.2: An Intention Structure

Of course, it is essential that, while waiting for an activation condition to become true, the system continue to monitor the environment and continue to execute other intentions as required. Thus, for example, while waiting for the pressure to drop below 312 psi, the RCS system keeps monitoring the status of the shuttle, responds to any observed changes in the situation (such as a failure in some other component), and performs any other tasks demanded of it.

3.6 Intention Types

One of the most interesting features of PRS is the manipulation of intentions within the intention structure. The purpose of this and the following two sections is to explain the mechanisms for manipulating the intention structure and how they can be customized to a particular application.

The intention structure contains those KAs that have been chosen by the system to be executed. The system commits to these intentions; it will "try its best" to achieve them and plan its other activities in accord with them. The set of intentions comprising the intention structure form a partial ordering such as shown in Figure 3.2. Those that are roots of this graph (i.e., have no predecessors) are candidates for becoming the *current intention.* The current intention is the one that is currently being executed, and is surrounded by two small arrows in the figure. The directional arcs shown in the figure represent precedence constraints on the intentions. That is, the intention earlier in the ordering (as defined by the arcs in the partial order) must be finished (and thus disappear from the intention structure) before intentions appearing later in the ordering. This precedence relationship between intentions enables the system to establish priorities and other relationships between intentions.

An intention can be in one of three possible states:

Normal: This is the most common state. If such an intention is a root of the graph then it can be activated. Otherwise, it must wait until all its predecessors are finished.

Sleeping: A sleeping (or conditional) intention is one whose execution is suspended, awaiting some condition to be satisfied. To enter this state, an appropriate metalevel KA must be utilized. This metalevel KA can be invoked by the intention that is to be suspended or by some other intention outside it. The sleeping state implies the presence of an *activation condition* for the intention. This activation condition is a logical expression evaluable in the environment of the intention. As long as it is false, the intention is kept sleeping. As soon as it becomes true, the intention is put into a *woken* state.

Woken: As its name implies, an intention is in a woken state if has been "awakened" after having been in a sleeping state. The woken state is exactly the same as the normal state, except that if there is more than one possible current intention, the system will prefer the most recently woken one. As a result, the system will tend to activate intentions which have just been awakened. As soon as such an intention has been activated, the woken state is exited and the intention returns to the normal state.

3.7 Establishing Intentions — The System Interpreter

Intentions are established in two ways: (1) by the system interpreter; and (2) by particular metalevel KAs. The operation of the interpreter in establishing intentions is worth examining. The main problem to be solved is that, on any cycle, a number of KAs may be applicable. It is thus necessary to decide what to do with these applicable KAs — in particular, how many (if any) to establish as intentions and how to insert those so chosen into the intention structure. The notion of metalevel KAs was introduced to provide maximum flexibility in making these decisions.

But we have to find a mechanism for bringing these metalevel KAs to bear at the appropriate time. The way we chose to do this was to include in the invocation part of the metalevel KA some condition on the number or kind of object-level KAs that are applicable at each cycle. For example, a particular metalevel KA might be invoked on the basis of there simply being more than one applicable object-level KA at the current moment.

To enable this scheme to work, we first have to determine which object-level KAs are applicable on each cycle. This information becomes a new system belief. In particular, on each cycle, the system acquires a belief about the set of KAs applicable on that cycle, expressed as (**soak** x), where x is the list of applicable KAs. It is then determined whether or not the aquisition of this new belief (i.e., (**soak** x)) triggers any new [metalevel] KAs. If it does, the system acquires a new belief about the applicability of these metalevel KAs. In fact, it does so simply by updating the belief (**soak** x) so that the list x now contains exactly those metalevel KAs that are now applicable. (The previous belief about applicable object-level KAs is removed from the database and so, in a sense, is forgotten. However, it is captured in the variable bindings of the invoked metalevel KAs.)

As PRS places no restrictions upon the invocation conditions of metalevel KAs, it is quite possible that more than one metalevel KA will be invoked at this stage. If this happens, we will now be left with the problem of deciding which of these metalevel KAs to invoke. There are a number of possible solutions to this problem. One would be simply to select one of the metalevel KAs at random, on the assumption that all are equally good at making the decision about which object-level KAs should be invoked. Another alternative would be to preassign priorities to the metalevel KAs and to invoke the one with the highest priority. However, in keeping with our aim of providing maximum flexibility, the solution we chose to adopt is to allow further metalevel KAs to operate on these lower-level metaKAs in the same way that the lower-level metaKAs operated on the object-level KAs.

The process of invoking metalevel KAs is thus continued until no further KAs are triggered.[3] At that point, there may still be a set of applicable KAs from which to choose. It is then, and only then (i.e., only after failing to find any more applicable metalevel KAs), that we select one of these KAs at random.

Thus it is seen that, when more than one KA is applicable, and in the absence of any information about what is best to do, the system interpreter defaults to selecting one of these KAs at random. With no metalevel KAs, the system would thus randomly select one of the applicable object-level KAs. However, one usually provides metalevel KAs to help make an informed choice about the object level KAs. The applicable metalevel KAs themselves are subject to the same default action (i.e., one will be randomly selected) unless there are yet other metalevel KAs available to make a choice among them. In the end, at some level in the meta-hierarchy, the default action will be taken (of course, there may, at that level, be only one KA to choose).

Once selected, the chosen KAs must be inserted into the intention structure. If a selected KA arose due to an *external goal* or a *fact*, it will be inserted into the intention structure as a new intention at the root of the structure. For example, this will be the case for any metalevel KA that is invoked to decide among some set of applicable lower-level KAs. Otherwise, the KA instance must have arisen as a result of some subgoal of some existing intention, and will be "grown" (i.e., attached) as a subKA of that intention.

[3]Currently, it is left to the user to ensure that the triggering of higher and higher levels of metaKAs terminates in a bounded time. This is not difficult for the user to achieve. For example, one could assign each metalevel KA to one of a finite set of fixed levels (types), and enforce the condition that KAs only operate on others at the level below them in this finite hierarchy.

selector2

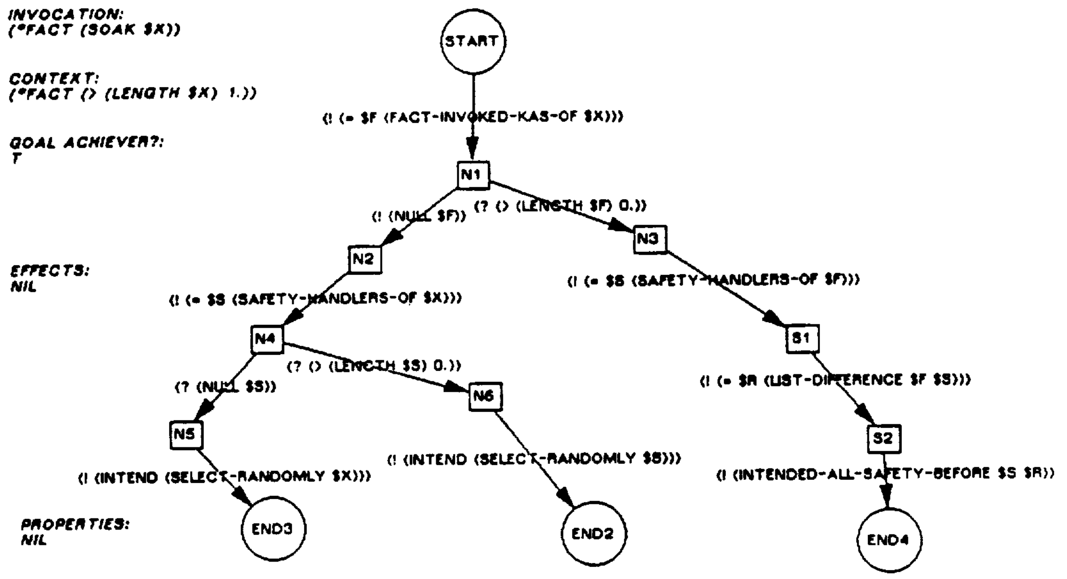

INVOCATION:
(*FACT (SOAK $X))
CONTEXT:
(*FACT (> (LENGTH $X) 1.))
GOAL ACHIEVER?:
T
EFFECTS:
NIL
PROPERTIES:
NIL
START
(! (= $F (FACT-INVOKED-KAS-OF $X)))
N1
(! (NULL $F))
(? (> (LENGTH $F) 0.))
N2
N3
(! (= $S (SAFETY-HANDLERS-OF $X)))
(! (= $S (SAFETY-HANDLERS-OF $F)))
N4
S1
(? (NULL $S))
(? (> (LENGTH $S) 0.))
(! (= $R (LIST-DIFFERENCE $F $S)))
N5
N6
S2
(! (INTEND (SELECT-RANDOMLY $X)))
(! (INTEND (SELECT-RANDOMLY $S)))
(! (INTENDED-ALL-SAFETY-BEFORE $S $R))
END3
END2
END4

Figure 3.3: A Metalevel KA

3.8 Manipulating Intentions — Metalevel KAs

The only other way to establish intentions is by invoking special metalevel KAs. These provide a variety of options for inserting KAs into the intention structure.

There are four different ways in which an intention can be inserted into the intention structure:

1. As a subKA of the current intention.
2. As a new intention at the root of the intention structure; i.e., prior to every other intention, including the current intention if it has not finished execution.
3. As a new intention that has *precedence* over some set of existing intentions.
4. As a new intention that will be initiated only *after* some set of existing intentions have been executed.

For example, consider the metalevel KA `selector-2` in Figure 3.3. The goals `(!(intend ...))` and `(!(intend-all-safety-before ...))` are satisfied by metalevel KAs that eventually establish intentions, each in its own way. In particular, the KA that responds to the first goal inserts the KA instance directly after the current intention, whereas the KA that responds to the second goal gives those object-level KAs that have the property of being "safety handlers" priority over those that are not.

To construct sophisticated metalevel KAs, it is usually necessary to provide additional information about KAs – for example, which are "safety handlers" or which are fact invoked (see Figure 3.3). As mentioned previously, these properties can be stored in the system database or, more conveniently, directly represented in the appropriate slots of the KA structures.

Finally, it is important to be careful in the use of metalevel KAs. In particular, it is sometimes quite difficult to foresee when the invocation condition of a metalevel KA will become true. For example, consider the following activation condition, where **soak** represents the set of KAs that are currently applicable:

```
(and (*fact (soak $x))
     (*fact (equal 1 (length $x))))
```

This condition will be true whenever there is exactly one applicable KA (i.e., the set of applicable KAs, **$X**, contains only one member). A KA with such an invocation condition will possibly loop on itself indefinitely! This is because the KA will repeatedly become the single applicable KA, thus reactivating itself.

The initiating events for some of the more important metalevel KAs are listed below. Some of these KAs are used to create new intentions, others to modify the intention structure itself, and yet others to modify the status of exisiting intentions.

Intend the KA instance **$x**:
Initiating Event: `(*goal (!(intend $x)))`

Intend all KA instances in the set **$x**, but give priority to "safety handlers":
Initiating Event: `(*goal (!(intend-all-safety-before $x)))`

Make the current intention sleep for **$t** seconds:
Initiating Event: `(*goal (!(sleep $t)))`

Awake a sleeping intention:
Initiating Event: `(and (*fact (wake-up $x)) (*fact (> (length $x) 0)))`

Wait until the condition **$c** is true (the current intention is automatically put to sleep):
Initiating Event: `(*goal (!(wait-until $c)))`

3.9 Commitment to Intentions

PRS commits to its intentions. That is, unless some particular metalevel KA intervenes, PRS will perform its means-ends reasoning and other planning *in the context of its existing intentions.* For example, consider that PRS has adopted the intention of achieving a goal *g* by accomplishing the subgoals g_1, g_2, and g_3, in that order. In the process of determining how to accomplish these subgoals, the system will not reconsider other means of achieving g. That is, it is *committed* to achieving g *by* doing g_1, g_2, and g_3, even if circumstances have so changed that there is now a better way to achieve g than the one chosen. The gain here is in reducing decision time — in highly dynamic domains it is not possible to continually reassess one's plans of action. What makes the approach workable is that the basis upon which one chooses a particular plan of action is more often correct than not.

Of course, the system is not committed to its intentions forever. For example, as discussed in Section 3.3, if PRS determines that it cannot achieve g by doing g_1, g_2, and g_3, it will drop that plan and look for some other means of achieving g. Alternatively, it may remember the basis for choosing one plan over another, and utilize appropriate metalevel KAs to modify its intentions if support for that decision is subsequently found to be lacking.

It is not only in means-ends reasoning that PRS's commitment to its existing intentions is important. For example, in tackling some new task, it is often desirable that the means or time chosen for accomplishing that task take account of one's existing intentions towards the fulfillment of other tasks. In the space-shuttle application, this happens, for example, when the PRS instance `INTERFACE` receives a request for a pressure reading when it is in the process of evaluating the status of a suspected faulty transducer. In this case, `INTERFACE` will either defer or suspend attention to that request (possibly advising the requester) until it has completed its evaluation of the transducer.

3.10 Supporting Goals and Beliefs

Some embedded reasoning systems are designed so that any given course of action is terminated whenever the beliefs or goals that brought about that action cease to be true. In this section, we shall consider this approach and how it compares to the one adopted in PRS.

Suppose that a certain alarm sounds, which in turn invokes some procedure to rectify the fault. In some cases, the alarm will only be turned off when the problem is rectified. Thus, it appears sensible that, should ever the alarm cease to sound, any procedure aimed toward that end be terminated (but more about this below). On the other hand, there are many cases where the ceasing of an alarm does not mean that the corresponding fault has been fixed: the alarm may only sound for a fixed interval of time, or it may be deliberately turned off to relieve the operator of the annoying noise. Thus, it is clearly bad policy to *always* terminate a procedure simply on the grounds that the beliefs that caused its activation no longer hold. Of course, if the inititiating belief is that the alarm sounded *at a particular time*, and *this* belief is found to be false, it is usually sensible not to proceed any further.

Should the goals that constitute the purpose of some course of action cease to exist, the rationale for continuing with the action disappears also. However, this need not mean that one should simply cease doing what one was doing and get on with other activities. Indeed, in most cases one would need to "clean up" after the action, and in some cases one may even desire to see things through to the end (as, for example, when one has fired the jets to force reentry of the shuttle).

Thus, depending on circumstances, one has to reason carefully about the early termination of procedures, even when there is due cause for termination. PRS offers no special techniques for handling these problems, but instead provides the hooks by which such reasoning can be performed *at the appropriate time.*

PRS will always recognize goal failures. Without any meta-KAs to respond to such failures, the system will simply try some other procedure to achieve the failed goal, eventually terminating once all avenues of attack have been explored. On the other hand, should one wish some special action to occur on goal failure, it is straightforward to construct appropriate metalevel KAs to perform the necessary corrective actions.

PRS currently has no notion of supporting beliefs for given courses of action. Thus, if it is desired that a certain procedure be terminated should some belief be no longer held, a special KA must be set up to look for that condition during the execution of the procedure. While this is not difficult to do, it would be useful to provide generic mechanisms to assist in this task. We plan to address this and related issues in our future work on PRS.

3.11 Guaranteed Reactivity

Definitions of real-time systems revolve around the notion of *response time.* For example, Marsh and Greenwood [11] define a real-time system as one that is " predictably fast enough for use by the process being serviced" and O'Reilly and Cromarty [12] require that "there is a strict time limit by which the system must have produced a response, regardless of the algorithm employed." This measure is most important in real-time applications; if events are not handled in a timely fashion, the operation can go out of control. Amazingly, few of the existing real-time AI systems are guaranteed to respond within a bounded interval of time [10].

Response time is the time the system takes to recognize and respond to an external event. Thus, a bound on *reaction time* (that is, the ability of a system to recognize or notice changes in its environment) is a prerequisite for providing a bound on response time. PRS has been designed to operate under a well-defined measure of reactivity. Because the interpreter continuously attempts to match KAs with any newly acquired beliefs or goals, the system is able to notice newly applicable KAs after every primitive action it takes.

To estimate the bound on reaction time, let p be an upper bound on the execution times of the primitive actions that the system is capable of performing. Let's also assume that n is an upper bound on the number of events that can occur in unit time, and that the PRS interpreter takes at most time t to select the set of KAs applicable to each event occurrence.[4]

Thus, the maximum *reactivity delay*, Δ_R, is given by

$\Delta_R = p + y \times t$

where y is the maximum number of events that can occur during the reaction interval.

We have

$y = \Delta_R \times n$

and thus we obtain

$\Delta_R = p/(1 - nt)$

where we assume that $t < 1/n$.

This means that, provided the number of events that occur in unit time is less than $1/t$, PRS will notice *every* event that occurs [that is capable of triggering some KA] and is guaranteed to do so within a time interval Δ_R. In the current implementation, the values of p and t are less than 0.1 seconds, giving a reactivity delay of at most 0.2 second for an event rate of 5 events per second.

Because metalevel procedures are treated just like any other, they too are subject to being interrupted after every primitive metalevel action they take. Thus, reactivity is guaranteed even when the system is choosing between alternative courses of action or performing deductions of arbitrary complexity.

Having reacted to some event, it is necessary for the system to respond to this event by performing some appropriate action. As the system can be performing other tasks at the time the critical event is observed, a choice has to be made concerning the possible termination or suspension of those tasks while the critical event is handled. Furthermore, if there are a number of different ways in which the event can be handled, it is also necessary to choose among those alternatives.

Such choices can be made by appropriate metalevel KAs. However, in general, these decision procedures may take an unbounded amount of time to reach a decision. There are two possible ways to overcome this problem. One is to require that all decision procedures complete in a bounded time. In many domains, this provides adequate decision-making capability and yields a bound on response time.

Alternatively, one could construct a special metalevel KA to act as a task scheduler. This KA would have the capability to preempt all executing decision tasks (and any other tasks for that matter) within a bounded time and begin execution of an event handler. It could utilize whatever information was available (such as any incremental decisions made in

[4]As selection of KAs does not involve any general deduction beyond unification and evaluation of a boolean expression, an upper bound does indeed exist.

narrowing down the range of possibilities) to select the most appropriate event handler and the manner in which to suspend or terminate other tasks. It could also take into account the different constraints on response time that may exist in different situations. The only requirement is that this KA have a guaranteed upper bound on execution time.

In summary, PRS is guaranteed to react to critical events in a bounded time interval. With appropriate metalevel KAs, it is also possible to guarantee a bound on response time. However, the question remains as to what algorithms (KAs) are well suited for decision-making in bounded time. Surprisingly, it is only recently that researchers in AI have begun to consider this problem [2, 4]. We propose to address this issue more fully in our future research.

3.12 Variable Usage

The design of the PRS system has attempted to stress the use of logical variables, similar to the way variables are used in Prolog. Within a KA, such variables can never be rebound. However, it is sometimes convenient to use variables with other kinds of semantics, in particular, to use variables that have a semantics similar to what is found in standard programming languages.

There are three kinds of variables in the PRS system, all of whose semantics can be mapped onto a standard logical variable semantics, but whose usage differs. *Global variables* are just like logical variables in the classic sense. *Local variables* are like global variables, but have a limited extent or lifetime. *Program variables* function much like normal variables in standard programming languages.

Global variables are prefixed by `$` and can have only one binding during the lifetime of a particular KA instance.[5] Note that each instantiation of a KA is associated with its own global variables. Thus, the global variables in each recursive call of the same KA will be distinct.

Local variables are prefixed by `%`; they behave like global variables, except that their binding is meaningful only on a single arc. In other words, on each arc, the appearance of a `%` variable is similar to the creation of a fresh new global variable. The binding of a `%` variable will *not* carry over from one arc to the next. However, if multiple goals are invoked on the same arc, the same variable and binding will be used for all of them.

Program variables are prefixed by `@`; they can be rebound from arc to arc and retain their value between arcs. Whether or not the value of an `@` variable is rebound or not will depend upon context. A goal of form `(!(= @x @y))` will, by default, bind the value of `@x`

[5]Of course, as in Prolog, many different bindings may be attempted in trying to satisfy some goal expression that appears in the KA. However, once a binding has been found that successfully achieves that goal, that binding is fixed for the lifetime of the KA instance.

to be equal to the current value of `@y`, if it has one. If `@y` was *not* bound and `@x` was bound when this goal occurred, `@y` would be bound to the value of `@x`. If both were unbound, they would nevertheless be constrained to be equal, and if one achieved a binding later, the other would be bound to that binding as well. A goal of form `(?(= @x @y))` simply tests to see if the bindings of the two variables are equal.

Clearly, both global and local variables are *logical* variables in the classic sense – they cannot be rebound. In order to handle the semantics of `@` variables consistently within this logical-variable framework, the binding of program variables may be viewed as a particular property of an associated global variable (intutively, the contents of the denoted object). Whereas global variables can be bound only once, it is acceptable for the value of its *contents* (a *function* on the variable) to change. For example, one could imagine a global variable as a box. Each global variable is a specific box and can never be rebound to another box, but different objects may be placed *in* the box at different points in time.

Note that variables are assumed to bind to so-called *rigid* designators – that is, the variable is assumed to denote the same object throughout the KA in which it appears. Thus, any functional expressions (which could be bound to such variables) should also be rigid designators. In such cases, one should not, for example, use a function such as `(the-block-on-top-of $x)` as the denotation of this expression could well change during the execution of a KA. This restriction does not apply, however, to evaluable functions, as they and their arguments are immediately evaluated by the system.

3.13 Planning or Not?

There has always been some confusion in the literature about the notion of planning, especially with respect to the kind of practical reasoning that PRS performs.

In the AI literature, planning is viewed as the generation of a sequence of actions to achieve some given goal. The classical approach to this problem is to simulate the effects of performing the actions so as to ensure that their execution does indeed achieve the required goal. All this planning is done, in most cases, prior to performing any physical action in the actual world.

It is quite straightforward to run PRS in this way: the primitive actions performed by the system are decoupled from the actual world, and the KAs simply become the "operators" of classical planning systems. Thus, the system simulates execution of the KAs, and its database reflects beliefs about the state of the world as it would be had those KAs actually been executed. As the system explores all possible sequences of activity that could possibly lead to the goal condition being achieved, it will find a plan if one exists.[6] In this sense, PRS is capable of planning in the classical tradition, albeit not very efficiently.

[6]In fact, this depends on how the metalevel KAs are written. In particular, one has to ensure that all possible interleavings of any conjunctive goals are explored.

The kind of planning discussed above takes place prior to performing any actions in the actual world. However, it is also possible to form plans during the course of performing some task. Assume one has some goal, g, and a variety of ways to achieve that goal. Let's say that there are two options: achieve g_1 followed by g_2; or achieve f_1 followed by f_2. Now one could choose arbitrarily between these options, or one could engage in some level of planning to determine which was the best course of action in the given circumstances. This kind of planning may involve simulating the possible outcomes of each approach by elaborating these options as done in classical planning. However, one could alternatively select from a great variety of other techniques. For example, the choice could be based on the expected time to complete the actions, or the likelihood of success of the plans as gained through experience. In any case, simply choosing which course of action to pursue, no matter how one does it, constitutes forming a *plan* to achieve the goal g. Having chosen one of these courses of action (or, indeed, none[7] or both![8]), one repeats the process. For example, if the course involving g_1 and g_2 were chosen, and one had various ways of achieving g_1, then it would be necessary to plan how best to achieve that subgoal, and so on.

This is exactly the way PRS operates. The method of choosing between alternative courses of action is embedded in the metalevel KAs of the system and thus, in essence, the particular *approach* to forming plans is not hard-wired into the system. To the extent that the choice is made arbitrarily, one may wish to avoid calling this process "planning." But where it is based on any information at all, no matter how meager, the determination of an appropriate course of action is indeed planning.

In the RCS example discussed in Chapter 4, the system decides between different courses of action depending on how the KA was invoked and what sort of priority it has. This is clearly quite a weak form of planning, and more complex meta-KAs – taking time availability, costs, and benefits into account – would improve system reliability.

Of course, it is important to determine exactly what algorithms (metalevel KAs) are needed for effective planning. The RCS problem, as we said above, uses a very simple form of planning, which, in itself, is probably not of much interest. However, what is of interest is just how weak the planning component can be when we have a wealth of experience (a rich set of object-level KAs) to assist us.

3.14 Additional Features of PRS

This report is concerned with the design of real-time reasoning systems, and not primarily with features of the particular implementation of PRS at SRI International. However, it

[7]Such as when both approaches appear unlikely to achieve the goal.

[8]Such as forming the plan to court both Jane and Mary in parallel, in the hope that one of them will eventually marry you.

is important to mention some of the major implementation features that facilitate creating and modifying system databases and KA libraries.

3.14.1 The KA Editor

Each PRS system must include a set of KAs supplied by the user. Normally, KA descriptions are stored within a KA library. Each KA description includes a network of labeled nodes and arcs, as well as an invocation condition that describes the situations in which that KA is applicable and useful. The user of the PRS system inputs all of this procedural information using a *KA Editor*, which is based on SRI's proprietary **Grasper-II** system. This user interface is fairly straightforward and guides the user through the creation or modification of selected KAs via questions and menus.

To create a new KA, the user is prompted with a series of questions. The user will be asked to enter the invocation condition for this KA and various other information. After answering these questions, the user can begin creating the actual KA body (network) using the appropriate menu items. These allow the user to create and name nodes and arcs, to specify their shape, and to perform various other graphical operations on the KA. Once created, KAs can be easily modified using the same menu-driven system.

3.14.2 The Structure Editor

When applied to certain domains, knowledge-based inference systems must make use of large amounts of information relating to the physical structure of the real world. In these cases, encoding this information as predicates in a database is a critical and very time-consuming operation. The way this is normally done is by taking each physical object in the domain and writing down facts about its attributes (i.e., size, weight), class relationships (A is a brick), and relations to other objects in the domain (A is connected to B, A is inside C, etc.). This operation becomes very inefficient as the number of objects in the domain becomes large. The task of building the database can take a very long time, and any modification of the structure of an object requires an expensive search through the database to change all the facts that involve that object.

To overcome this problem, PRS employs a *Structure Editor* that allows the representation of PRS's predicate calculus database about structure in an interactive graphical format. As with the KA Editor, the Structure Editor is built on top of the Grasper-II system. In addition to making it easier to create and update the database, the Structure Editor also incorporates features of object-oriented programming (like classes and inheritance) that facilitate the creation of large and conceptually complex systems of objects. The Structure Editor can also be used to alter structure dynamically as the PRS system runs, and thus can display visually certain the current status of the application system.

3.14.3 The Natural Language Interface

SRI International's natural-language system, **Candide**, has been used to facilitate the building of KAs. Using Candide, an engineer or mission controller can readily create KAs through a mixed interaction involving both graphics and natural language. Candide translates English expressions into the logical form used by PRS, and operates in conjunction with the KA Editor described above.

Thus, using Candide, the user can specify complex objects, conditions, goals, and beliefs in English, their temporal and causal relations graphically. Candide also facilitates later modifications to the network via a natural-language dialog. Candide thus eliminates the need for those who design and maintain a PRS knowledge base to be proficient in the specialized logical language used by PRS. In addition, the use of English can greatly simplify what would otherwise be highly complex formal statements. For example, the entire portion of the procedural network that is highlighted in Figure 3.4 can be specified with the single query "Is the pressure in the affected manifold greater than 130?"

Significantly, Candide is not just a single-utterance interpretation system — it includes capabilities for processing the types of extended natural-language dialogues that are necessary in complex knowledge acquisition tasks. As one example, Candide tracks the discourse along each path of the network in order to perform sophisticated reference resolution. An engineer can specify an invocation constraint that "A jet is faulty." When the engineer later specifies that there should be a test for high usage in the RCS, Candide will determine that the intended RCS module is the one containing the faulty jet. Techniques for reasoning about the domain and the discourse history enable Candide to handle a range of important natural-language phenomena, including definite and indefinite reference, anaphora, quantifier scoping, resolution of nominal compounds, resolution of syntactic and lexical ambiguity, and metonymy.

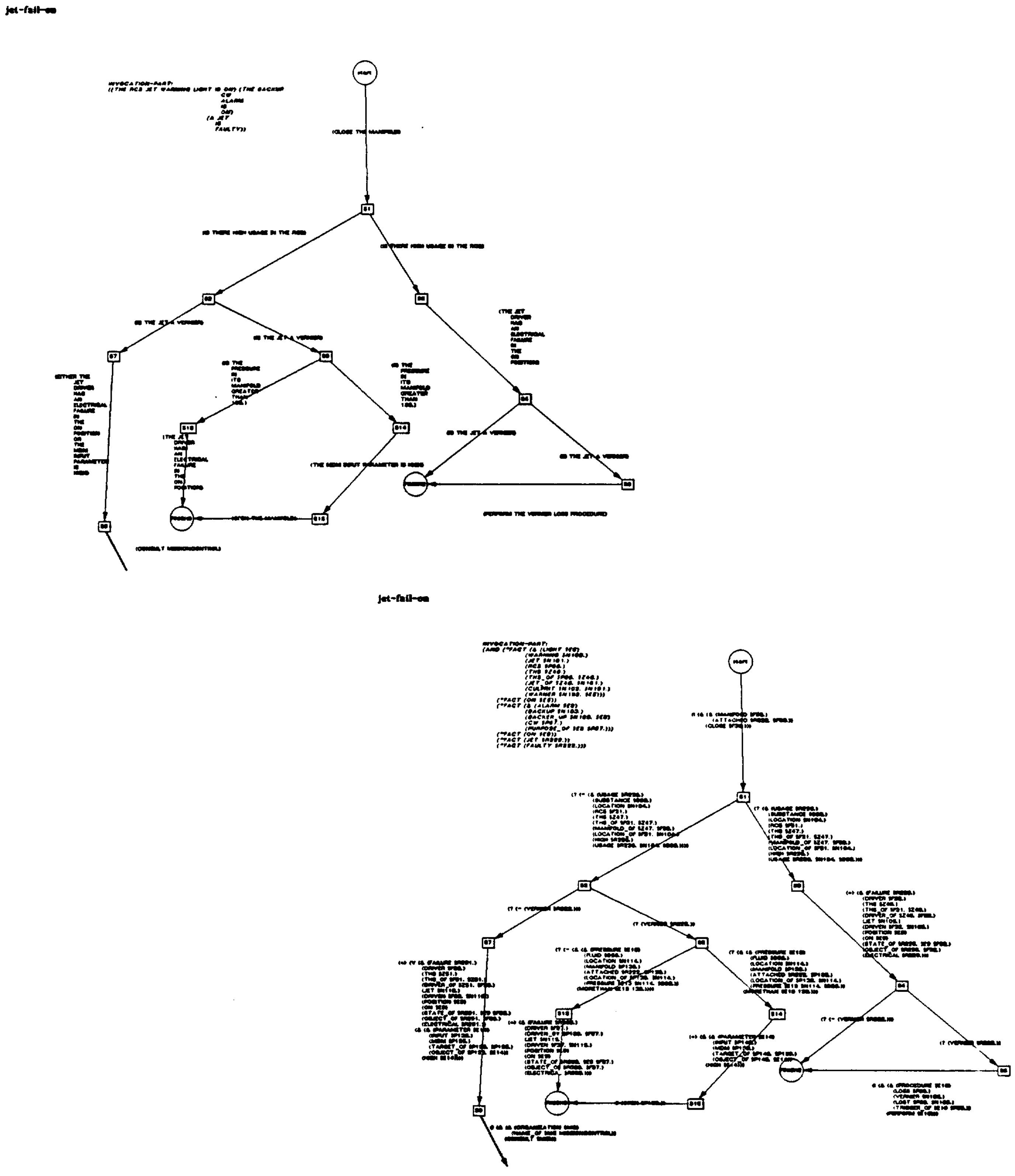

jet-fail-on
INVOCATION-PART:
((THE RCS JET WARNING LIGHT IS ON) (THE BACKUP CW ALARM IS ON) (A JET IS FAULTY))
(CLOSE THE MANIFOLD)
S1
(IS THERE HIGH USAGE IN THE RCS)
(IS THERE HIGH USAGE IN THE RCS)
S2
S3
(IS THE JET A VERNIER)
(IS THE JET A VERNIER)
S7
S8
(THE JET DRIVER HAS AN ELECTRICAL FAILURE IN THE ON POSITION)
(EITHER THE JET DRIVER HAS AN ELECTRICAL FAILURE IN THE ON POSITION OR THE MDM INPUT PARAMETER IS HIGH)
(IS THE PRESSURE IN ITS MANIFOLD GREATER THAN 150.)
(IS THE PRESSURE IN ITS MANIFOLD GREATER THAN 150.)
S4
S13
S14
(THE JET DRIVER HAS AN ELECTRICAL FAILURE IN THE ON POSITION)
(IS THE JET A VERNIER)
(IS THE JET A VERNIER)
(THE MDM INPUT PARAMETER IS HIGH)
FINISHED
S5
FINISHED
S15
(PERFORM THE VERNIER LOSS PROCEDURE)
S9
(CONSULT MISSIONCONTROL)
jet-fail-on
INVOCATION-PART:
START
S1
S2
S3
S7
S8
S4
S13
S14
S5
S15
S9
FINISHED
FINISHED

Figure 3.4: Example KA produced by Candide

Chapter 4

The RCS Application

The sytem chosen for experimentation with PRS is the reaction control system (RCS) of the space shuttle. The system structure is depicted in the schematic of Figure 4.1. One of the aims of our research is to automate the malfunction procedures for this subsystem. Sample malfunction procedures are presented in Figure 4.2 below, and in Chapter 2, Figure 2.1.

The RCS provides propulsive forces from a collection of jet thrusters to control the attitude and other motions of the space shuttle. Each jet is permanently fixed to fire in a particular direction. There are three RCS modules, two aft and one forward. Each module contains a collection of primary and vernier jets, a fuel tank, an oxidizer tank, and two helium tanks, along with associated feedlines, manifolds, and other supporting equipment. Propellant flow, both fuel and oxidizer, is normally maintained by pressurizing the propellant tanks with helium.

The helium supply is fed to its associated propellant tank through two redundant lines, designated A and B. The pressure in the helium tanks is normally about 3000 psi; this is reduced to about 245 psi by regulators that are situated between each helium tank and its corresponding propellant tank. A number of pressure and temperature transducers are attached at various parts of the system to allow monitoring.

Each RCS module receives all commands (both manual and automatic) via the space shuttle flight computer software. This software resides on five general purpose computers (GPCs). Up to four of these computers contain redundant sets of the Primary Avionics Software System (PASS) and the fifth contains the software for the Backup Flight System (BFS). All of the GPCs can provide information to the crew by means of CRT displays.

The various valves in an RCS module are controlled using a control panel of switches and talkbacks (Figure 4.3). Each switch moves associated valves in *both* the fuel subsystem and the oxidizer subsystem. Switches can be set to OPEN, CLOSE, or GPC, the last providing the GPCs with control of the valve position. The talkbacks provide feedback on valve position, and normally correspond with the switch position, except when the switch

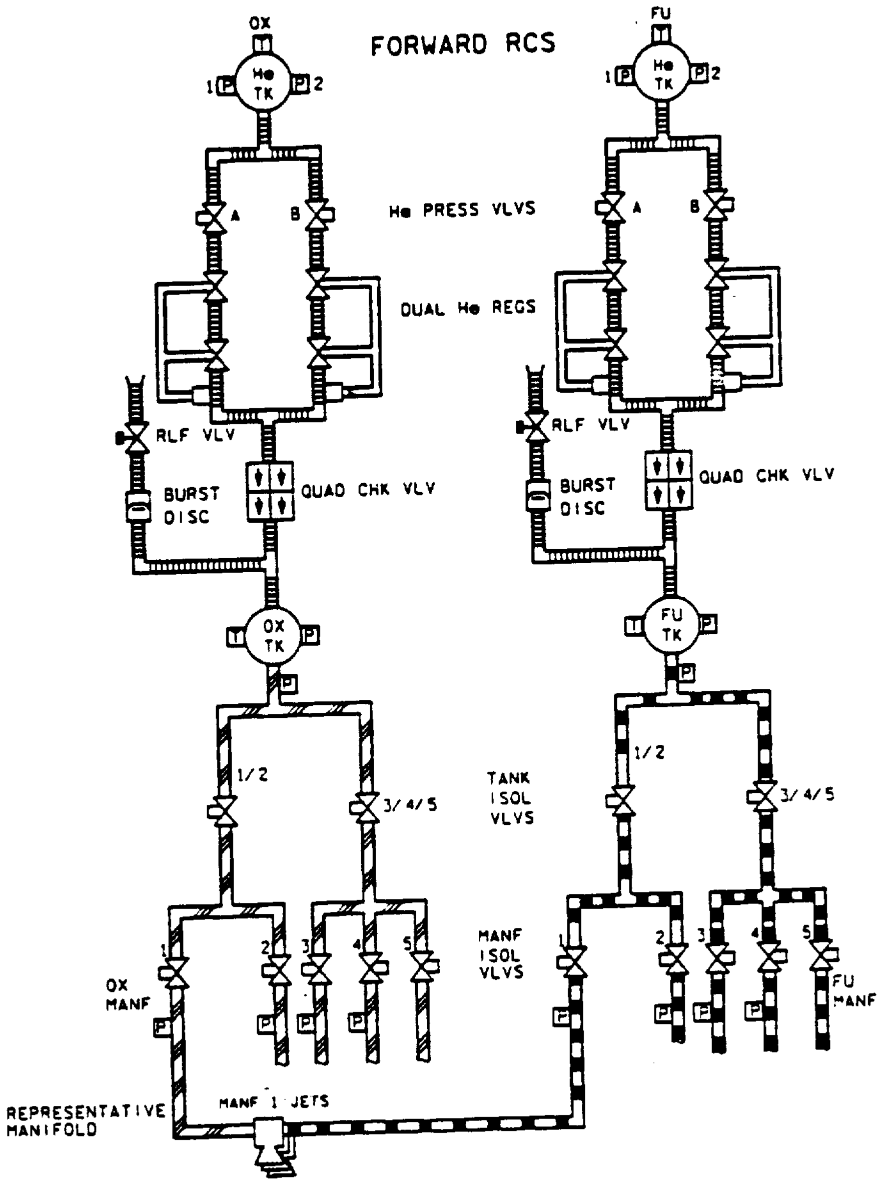


Figure 4.1: System Schematic for the RCS

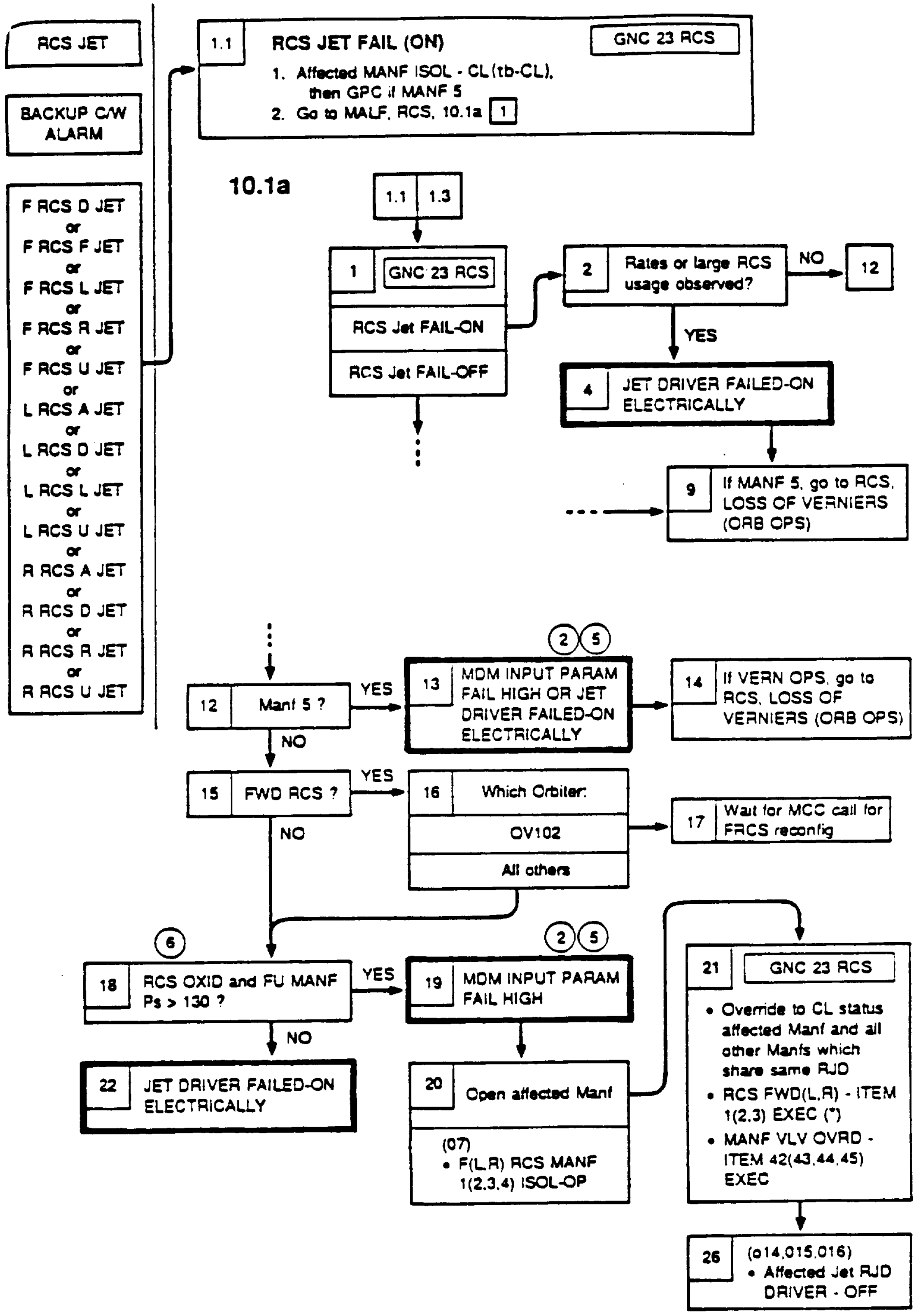

RCS JET
BACKUP C/W ALARM
F RCS D JET or F RCS F JET or F RCS L JET or F RCS R JET or F RCS U JET or L RCS A JET or L RCS D JET or L RCS L JET or L RCS U JET or R RCS A JET or R RCS D JET or R RCS R JET or R RCS U JET
1.1 RCS JET FAIL (ON) GNC 23 RCS
1. Affected MANF ISOL - CL (tb-CL), then GPC if MANF 5
2. Go to MALF, RCS, 10.1a 1
10.1a
1.1 1.3
1 GNC 23 RCS
RCS Jet FAIL-ON
RCS Jet FAIL-OFF
2 Rates or large RCS usage observed?
NO
12
YES
4 JET DRIVER FAILED-ON ELECTRICALLY
9 If MANF 5, go to RCS, LOSS OF VERNIERS (ORB OPS)
12 Manf 5 ?
YES
NO
2 5
13 MDM INPUT PARAM FAIL HIGH OR JET DRIVER FAILED-ON ELECTRICALLY
14 If VERN OPS, go to RCS, LOSS OF VERNIERS (ORB OPS)
15 FWD RCS ?
16 Which Orbiter:
OV102
All others
17 Wait for MCC call for FRCS reconfig
6
18 RCS OXID and FU MANF Ps > 130 ?
19 MDM INPUT PARAM FAIL HIGH
22 JET DRIVER FAILED-ON ELECTRICALLY
20 Open affected Manf
(07) • F(L,R) RCS MANF 1(2,3,4) ISOL-OP
21 GNC 23 RCS
• Override to CL status affected Manf and all other Manfs which share same RJD
• RCS FWD(L,R) - ITEM 1(2,3) EXEC (*)
• MANF VLV OVRD - ITEM 42(43,44,45) EXEC
26 (014,015,016) • Affected Jet RJD DRIVER - OFF

Figure 4.2: Some RCS Malfunction Procedures

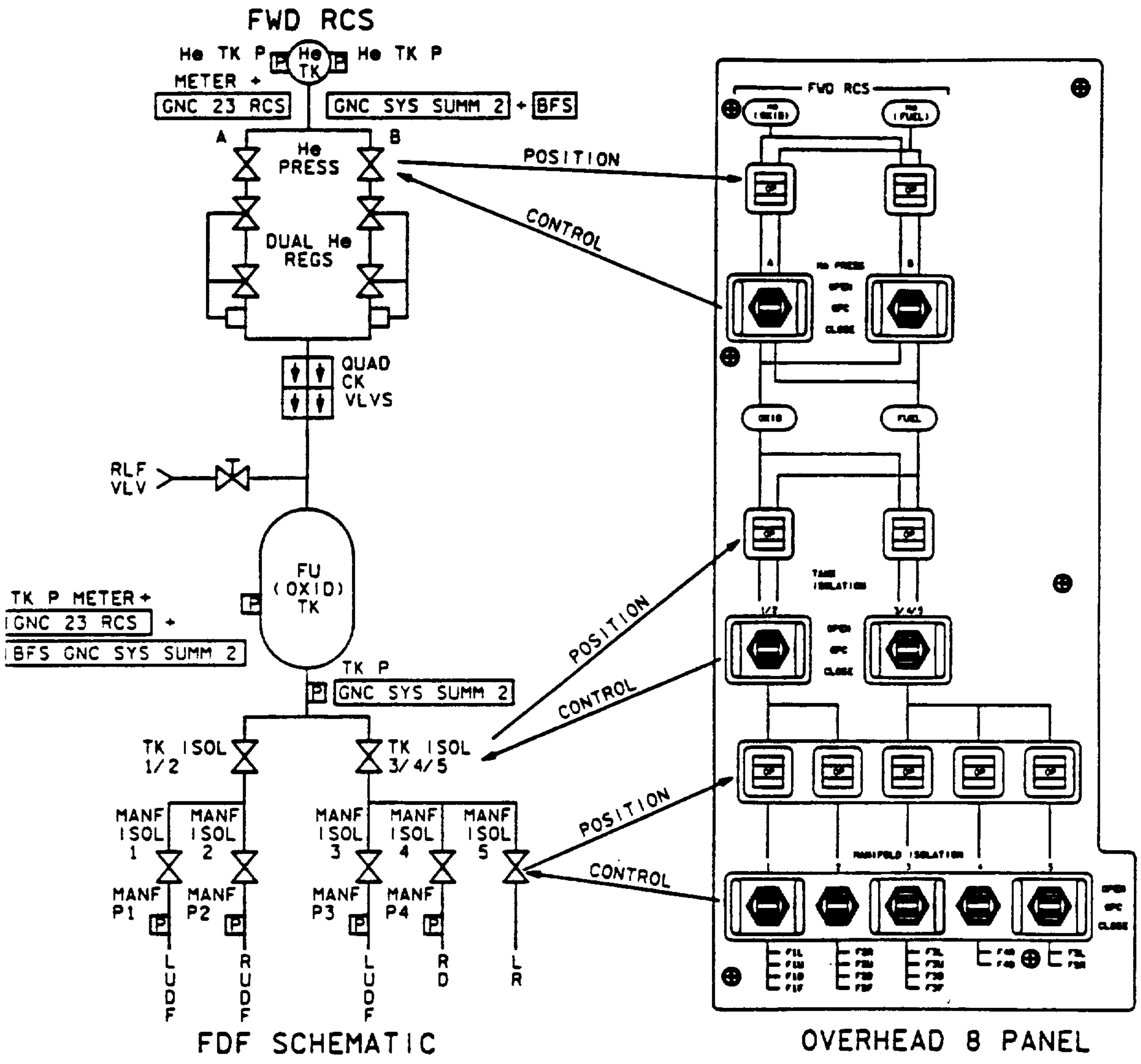


Figure 4.3: Controls for an RCS module

is in GPC. The talkbacks may not correspond if a valve has jammed or if the control or feedback circuit is faulty. If the valves in both the fuel and oxidizer subsystems do not move in unison (because of some fault), the talkback displays a barberpole.

It is important to note that, in the process of changing switch position, there will be a short time (about 2 seconds) when the positions of the talkback and the switch will differ from one another. This is because it takes this amount of time for the actual valve to change its position. Furthermore, during this transition, the talkback will also pass through the barberpole position. Thus, a mismatched talkback and switch position or a barberpole reading do not always indicate a system fault.

4.1 System Configuration

As mentioned earlier, two instances of PRS were set up to handle this application. One, called `INTERFACE`, handles most of the low level transducer readings, effector control and feedback, and checking for faulty transducers and effectors. The other, called somewhat misleadingly `RCS`, contains most of the high-level malfunction procedures, much as they appear in the malfunction handling manuals for the shuttle.

To test the system, a simulator for the actual RCS was constructed. During operation, the simulator sends transducer readings and feedback from various effectors (primarily valves) to `INTERFACE` and communicates alarm messages as they appear on the shuttle system displays to `RCS`. The simulator, in turn, responds appropriately to changes in valve switch positions as requested by `INTERFACE`. The simulator can be set to model a variety of fault conditions, including misreading transducers, stuck valves, system leaks, and regulator failures.

The complete system configuration is shown in Figure 4.4. Each of these is described in the following sections.

4.1.1 The Simulator

The simulator sends messages to `INTERFACE` and `RCS` about alarms, transducer readings, and talkback positions. In turn, it responds to commands from these systems to alter switch settings and reset alarms.

The simulator uses much the same predicates as used by `INTERFACE` for defining the configuration of the RCS and the various parameter values. A few additional predicates are used to fully specify the system and the status of the various system components.

Simulator Commands

The simulator can be set to simulate various conditions in the RCS. The following two commands simulate different modes of operation:

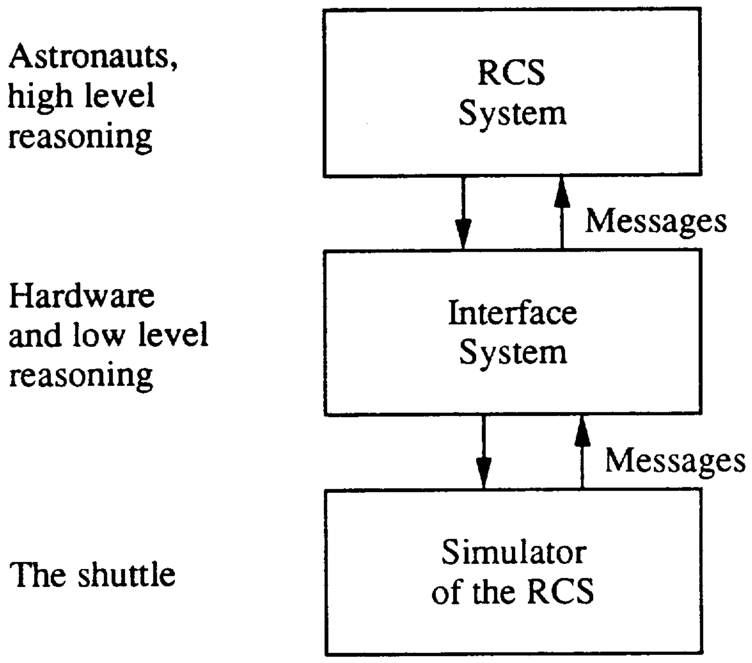


Figure 4.4: System Configuration

(OPENJETS) simulates system behavior when all 5 jets are firing.

(CLOSEJETS) simulates system behavior when none of the jets are using fuel or oxidizer.

The following interactive commands describe faults; these impinge on all future computations until removed.

(STATUS valve value) sets the status of valve. If BAD this particular valve will no longer change position; GOOD is normal.

(BROKEN-SWITCH valve) simulates a broken switch; the switch associated with valve no longer affects the position and talkback of its associated valves.

(BROKEN-TALKBACK valve) simulates a failure in the feedback of valve position; the talkback associated with this valve will remain stuck on its current value.

(BROKEN-XDCR xdcr value) simulates a broken transducer; the value of xdcr remains set to value; if value is not given then the value stays at its last value; if value is T the previous fault removed.

(LEAK xdcr message) simulates a leak in the tank, leg, or manifold associated with the transducer xdcr; it also triggers an alarm with value message.

(HELIUM-BLOCKAGE xdcr message) simulates a blockage in the helium pressure supply system; it triggers an alarm and does nothing else.

(BROKEN-REGULATOR regulator) simulates a broken regulator.

Transducer readings and valve and talkback positions can be set to arbitrary values by the following commands:

(VALUE xdcr n) sets the value of transducer xdcr to n and propagates the value.

(POSITION valve value) sets the position of valve to value, even if the valve switch is broken; talkback goes permanently to BP (barberpole) if the position of two valves controlled by the same switch differ.

(POSITION talkback value) sets talkback to value.

Of course, all commands described here are used for simulating different states of the RCS, and are beyond the control of both PRS instantiations and the astronaut or mission controller interacting with the system.

Simulator Input

The only input that the simulator can receive from either PRS instantiation is a command to change the position of a valve switch. This is achieved by the following command:

(POSITION switch value) sets switch to value, together with all associated valves and talkbacks.

In this application, only INTERFACE has the capability to command the simulator to change the position of a valve switch.

Each time the switch position is changed, the associated talkback goes through to that position in a time interval given by the variable *BARBERPOLE-INTERVAL*. This is normally set to about 2-4 seconds. While moving from the old position to the new position, the talkback is set to the barberpole setting. The positions of the corresponding valves are changed when the talkback reaches to its final position. If either the valve or talkback is faulty, the readings are fixed as described above.

Simulator Output

The simulator will send messages to INTERFACE whenever transducer readings or valve talkback positions change. This is done with the messages **(VALUE xdcr integer)** and **(POSITION talkback pos)**, respectively. The value of pos can be either OP (open), CL (closed), or BP (barberpole). Alarms are fired when given upper and lower limits are crossed, or certain leaks or blockages occur. All alarms are sent to RCS. Note that while the simulator holds information about actual pressures and valve positions, neither INTERFACE nor RCS are informed of these facts – they are simply informed of transducer readings and talkback positions, either of which could be in error.

Pressure Calculations

In this section we briefly describe the method for calculating pressures in the simulator. Each leak or firing jet is processed individually, incrementally changing all pressures in regions that are accessible through valves to the given leak or jet. Only after all these are processed does the simulator check final values for changes that must be reported.

Processing is done up to the *root* of the pressurization system by following ancestor links through open valves, then visiting all descendants of the root accessible via open valves, and updating their values on the way. The rates of fuel and oxidizer loss during jet firing, regulator pressure loss, and leakage are specified by *FIRE-RATE*, *REGULATOR-LEAK-RATE*, and *LEAK-RATE*, respectively. If the root is the helium tank, then the helium tank continues to pressurize the system despite the leak. We consider this case below. If the connection to a helium tank is newly opened, then all regions connected to it must allow the pressure to jump immediately to either *REGULATOR-PRESSURE* or the helium tank pressure, whichever is the less. If the pressure in these regions is already higher than either of these readings, then it remains unchanged.

We describe now how we approximate the loss of pressure in the system when the helium tank is pressurizing the leaking component. Leaks cause all regions accessible to them to lose pressure. When the helium tank is accessible to the leak, it continues to try to pressurize the system. We approximate this by having pressures decrease at some less-than-nominal rate, as specified by *HELIUM-LEAK-FACTOR*. We assume that the helium tank also loses pressure at this rate. Jet firings do not cause any loss of pressure when the system is pressurized by a helium tank.

When there is a leak on a helium tank, its pressure falls without affecting anything else until it gets down to the level of the fuel or oxidizer tank to which it is connected. At that point, the loss in pressure leak propagates to the rest of the system through all open valves.

Parameters and their Settings for the RCS Example

The values of variables used by the simulator are as follows:

`*LEAK-RATE* (-0.25)`: Rate at which pressure drops when a tank or leg leaks (psi/sec)
`*FIRE-RATE* (-0.1)`: Rate at which each jet uses pressure (psi/sec)
`*REGULATOR-LEAK-RATE* (1.0)`: Rate at which pressure increases if regulator fails (psi/sec)

`*P-THRESHOLD* (1.0)`: Minimum value of pressure increments (psi)
`*REGULATOR-PRESSURE* (245.0)`: Pressure maintained by regulator (psi)
`*TIME-DILATION* (1.0)`: Number of simulated seconds per actual second
`*BARBERPOLE-INTERVAL* (240.0)`: Time that talkback stays in barberpole, in sixtieths of a second
`*CYCLE-WAIT-TIME* (300.0)`: Time to wait at end of each cycle in sixtieths of a second
`*HELIUM-LEAK-FACTOR* (1.0)`: Relative leak rate of helium tank compared to other system leaks.

4.1.2 The RCS

The top-level PRS instantiation, **RCS**, contains most of the malfunction handling procedures as they appear in the operational manuals for the space shuttle. **RCS** takes an abstract view of the domain – it deals in pressures and valve positions, and does not know about transducers, switches, or talkbacks. For example, whenever **RCS** needs to know the pressure in a particular part of the system, it requests this information from **INTERFACE**, which is expected to deduce the pressure from its knowledge of transducer readings and transducer status. Similarly, **RCS** will simply request that **INTERFACE** move a valve to a certain position, and is not concerned how this is achieved. In this way, **RCS** can represent the malfunction handling procedures in a clean and easily understood way, without encumbering the procedures with various cross-checks and other details.

4.1.3 The INTERFACE

The PRS instantiation **INTERFACE** handles all information concerning transducer readings, valve switches, and valve talkbacks. It handles requests from **RCS** for information on the pressures in various parts of the system and for rates of change of these values. This can require examination of a variety of transducers, as readings depend on the status of individual transducers, their location relative to the region whose pressure is to be measured, and the connectivity of the system via open valves.

INTERFACE also handles requests from RCS to change the position of the valves in the RCS. This involves asking the astronaut to change switch positions, and waiting for confirmation from the talkback.

While doing these tasks, INTERFACE is continually checking for failures in any of the transducers or valve assemblies. When it notices such failures, it will notify the astronaut or mission controller, and appropriately modify its procedures for determining pressures or closing valves. It will also consider the consequences of any failures, such as are prescribed in various flight rules for the shuttle.

4.2 Sample Interactions

In this section, we briefly examine a few different scenarios.

4.2.1 Changing Valve Position

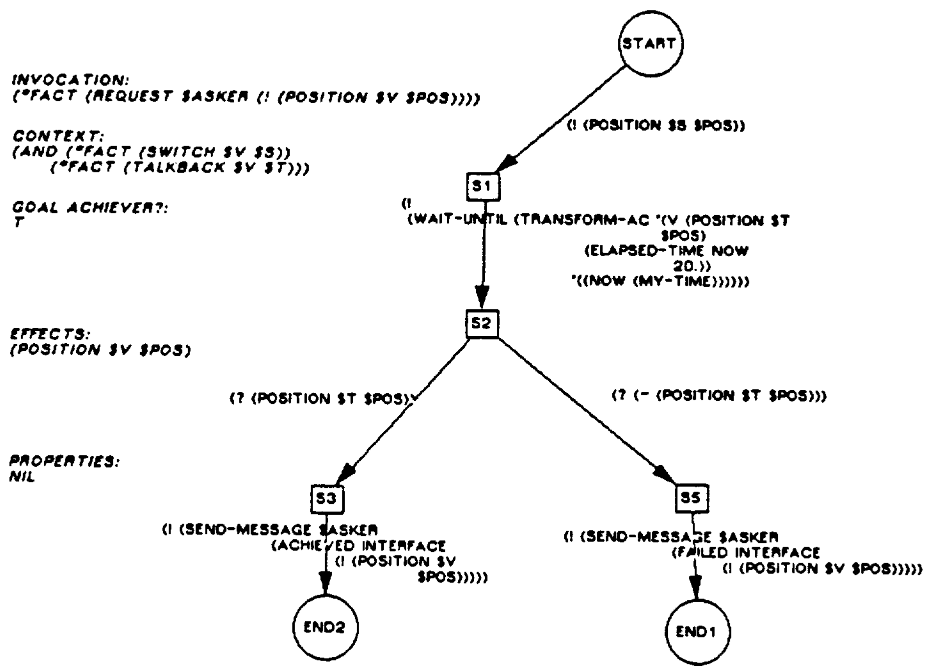
Let's consider the situation where INTERFACE gets a request from RCS to close some valve, say `frcs-ox-tk-isol-12-valve`. RCS achieves this by sending INTERFACE the message `(request RCS (!(position frcs-ox-tk-isol-12-valve cl)))`. Responding to this request, INTERFACE calls a KA that, in turn, asks the astronaut to place the switch corresponding to this valve in the closed position (see Figure 4.5). Once the astronaut has done this, INTERFACE will advise RCS that the valve has indeed been closed (Figure 4.5).

However, that is not the end of the story. INTERFACE will also notice that, just after the switch is moved to the closed position, there is a mismatch with the talkback indicator (which will still be showing open, due to the normal delay in the valve reaching its closed position). Furthermore, a fraction of a second later, the talkback will move into the barberpole position, another indication that things could be wrong with the valve.

Both these events initiate KAs that seek to confirm that the talkback moves to its correct position within a reasonable time (Figure 4.6). Each of these KAs or, more accurately, the intentions formed by these KAs, immediately suspends itself (using the "wait-until" goal) while awaiting a specified condition to become true.[1]

For example, the KA concerned with a possible switch dilemma will suspend itself until either the positions of both the switch and the talkback agree, or 20 seconds elapses. When either of these conditions become true, the KA (strictly, the intention) will awaken and

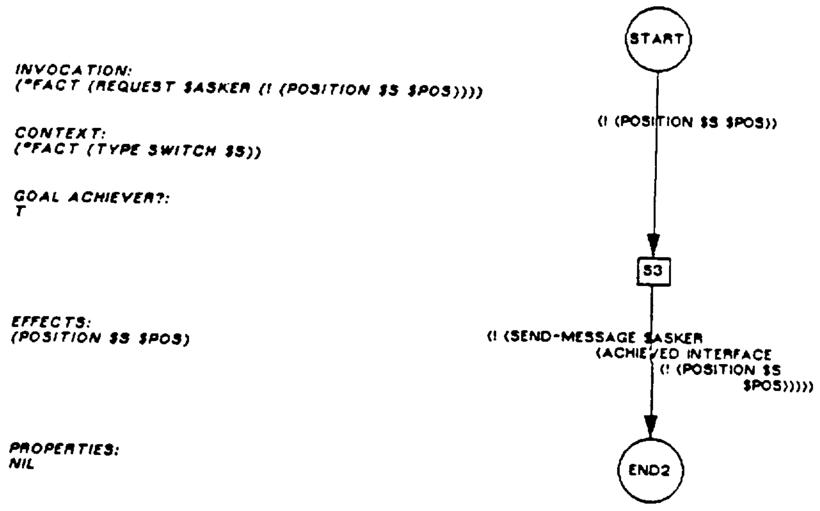
[1]The arguments to the wait-until goal are excessively cumbersome — for some important technical reasons — but can probably still be understood. Later versions of the system will allow a syntactically more convenient way of expressing the same information.



```
INVOCATION:
(*FACT (REQUEST $ASKER (! (POSITION $V $POS))))

CONTEXT:
(AND (*FACT (SWITCH $V $S))
     (*FACT (TALKBACK $V $T)))

GOAL ACHIEVER?:
T

EFFECTS:
(POSITION $V $POS)

PROPERTIES:
NIL
```

START
(! (POSITION $S $POS))
S1
(! (WAIT-UNTIL (TRANSFORM-AC '(V (POSITION $T $POS) (ELAPSED-TIME NOW 20.)) '((NOW (MY-TIME))))))
S2
(? (POSITION $T $POS))
(? (~ (POSITION $T $POS)))
S3
S5
(! (SEND-MESSAGE $ASKER (ACHIEVED INTERFACE (! (POSITION $V $POS)))))
(! (SEND-MESSAGE $ASKER (FAILED INTERFACE (! (POSITION $V $POS)))))
END2
END1

open-or-close-switch



```
INVOCATION:
(*FACT (REQUEST $ASKER (! (POSITION $S $POS))))

CONTEXT:
(*FACT (TYPE SWITCH $S))

GOAL ACHIEVER?:
T

EFFECTS:
(POSITION $S $POS)

PROPERTIES:
NIL
```

START
(! (POSITION $S $POS))
S3
(! (SEND-MESSAGE $ASKER (ACHIEVED INTERFACE (! (POSITION $S $POS)))))
END2

Figure 4.5: INTERFACE KAs for Closing a Valve

proceed with the next step. If the switch and talkback still disagree, it will notify the astronaut or mission controller of an error. Otherwise, it *fails*, and simply disappears from the intention structure.

Notice that the intentions to respond to the request from RCS, to monitor for a switch dilemma, and to check the barberpole reading are all established as intentions at some stage during this process. Various metalevel KAs must therefore be called, not only to establish these intentions, but to decide which of the active ones to work on next.

A typical state of the intention structure is shown in Figure 4.7. It shows a number of intentions in the system INTERFACE, ordered for execution as indicated by the arrows. The intention labeled soak is a metalevel KA. The other intentions include two that are checking potential switch problems and one that is responding to a request from RCS. The metalevel intention, in this case, is the one currently executing. Although not clear from the figure, it is trying to choose among a number of KAs that are applicable in the present situation.

4.2.2 Handling Faulty Transducers

In this case, we will assume that transducer frcs-ox-tk-out-p-xdcr fails and remains jammed at a reading of 170 psi. This causes a number of things to happen. First, it causes a low-pressure alarm to be activated which will be noticed by the PRS instantiation RCS. RCS will immediately respond to the alarm by initiating execution of the KA cw-rcs-alarm. This KA will, in turn, request a pressure reading from INTERFACE to ensure that the alarm is valid.

While this is happening, INTERFACE itself has noticed that the two transducers on the oxidizer tank do not agree with one another (in this case, the other transducer is reading the nominal value of 245 psi). This invokes the KA xdcr-bad, which attempts to determine which of the two transducers is faulty (Figure 4.8). It does this by first waiting a few seconds to ensure that the mismatch is not simply a transient, and then testing to see if one of the readings is outside normal limits. If so, it assumes this is the faulty transducer (this is indeed the procedure used by astronauts and mission controllers). Other KAs, capable of more sophisticated acts, such as checking the values of downstream or upstream transducers as well, could be added to the KA library if desired.

Notice what could happen here if one is not careful. Having more than one thing to do, INTERFACE could decide to service the request for a pressure reading for the suspect tank. If it does so, it will simply average the values of the two transducer readings (yielding 207 psi) and advise RCS accordingly. Clearly, this is not what we want to happen – any suspect parameter readings should be attended to before servicing requests that depend on them.

In the examples we have considered, it has been sufficient to handle such problems with a relatively simple priority scheme. We first ascribe the property of being a so-called

talkback-bp

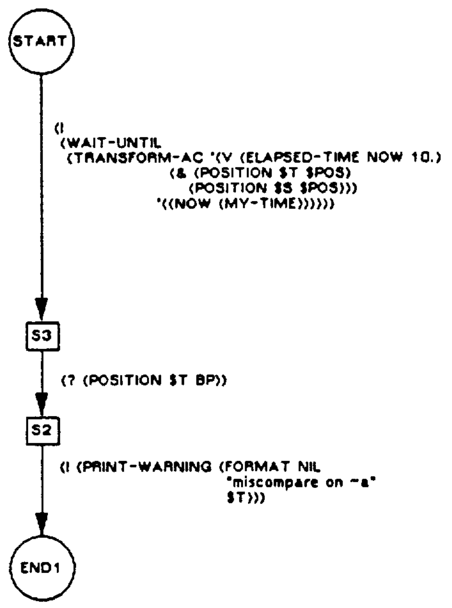
INVOCATION:
*(*FACT (POSITION $T BP))*

CONTEXT:
*(*FACT (ASSOCIATED-TALKBACK $S $T))*

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL



```
(!
 (WAIT-UNTIL
  (TRANSFORM-AC '(V (ELAPSED-TIME NOW 10.)
                  (& (POSITION $T $POS)
                     (POSITION $S $POS)))
                '((NOW (MY-TIME))))))
```

```
(? (POSITION $T BP))
```

```
(! (PRINT-WARNING (FORMAT NIL
                          "miscompare on ~a"
                          $T)))
```

switch-dilemma-1

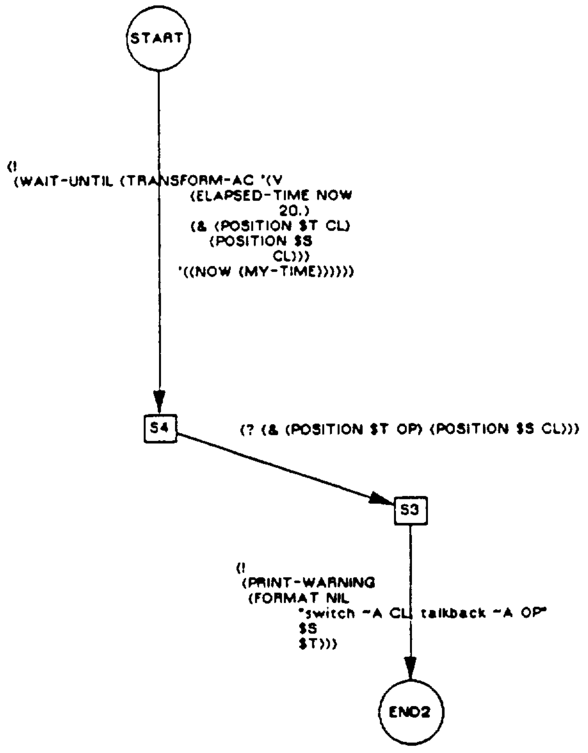
INVOCATION:
*(AND (*FACT (POSITION $T OP))*
*   (*FACT (POSITION $S CL)))*

CONTEXT:
*(*FACT (ASSOCIATED-TALKBACK $S $T))*

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL



```
(!
 (WAIT-UNTIL (TRANSFORM-AC '(V
                            (ELAPSED-TIME NOW
                                          20.)
                            (& (POSITION $T CL)
                               (POSITION $S
                                         CL)))
                          '((NOW (MY-TIME))))))
```

```
(? (& (POSITION $T OP) (POSITION $S CL)))
```

```
(!
 (PRINT-WARNING
  (FORMAT NIL
          "switch ~A CL talkback ~A OP"
          $S
          $T)))
```

Figure 4.6: INTERFACE KAs for Monitoring Valve Position

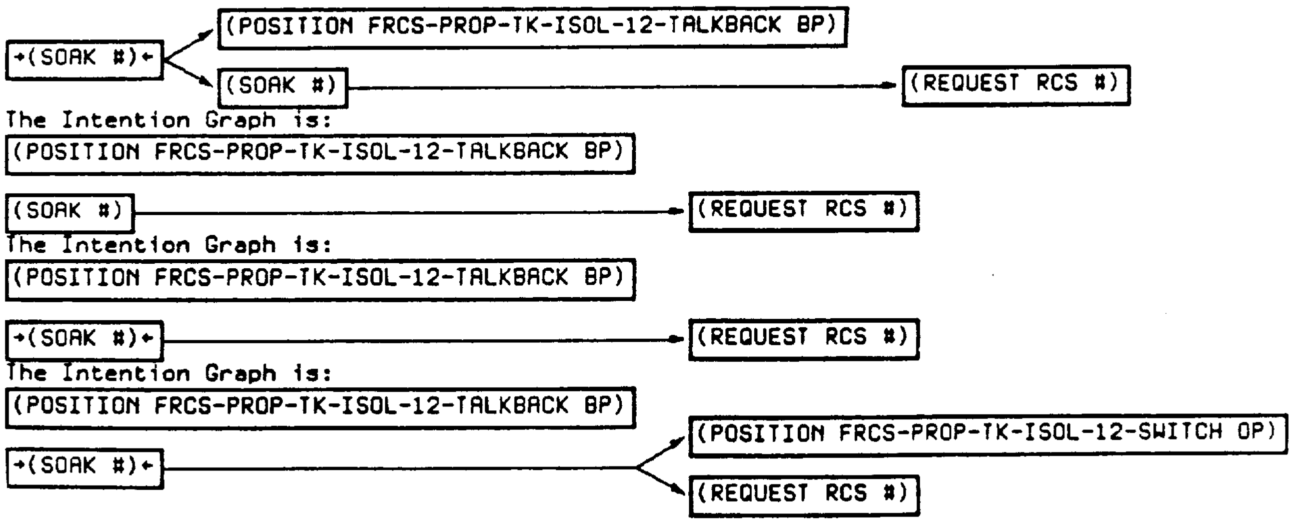

→(SOAK #)←
(POSITION FRCS-PROP-TK-ISOL-12-TALKBACK BP)
(SOAK #)
(REQUEST RCS #)
The Intention Graph is:
(POSITION FRCS-PROP-TK-ISOL-12-TALKBACK BP)
(SOAK #)
(REQUEST RCS #)
The Intention Graph is:
(POSITION FRCS-PROP-TK-ISOL-12-TALKBACK BP)
→(SOAK #)←
(REQUEST RCS #)
The Intention Graph is:
(POSITION FRCS-PROP-TK-ISOL-12-TALKBACK BP)
→(SOAK #)←
(POSITION FRCS-PROP-TK-ISOL-12-SWITCH OP)
(REQUEST RCS #)

Figure 4.7: Typical Intention Structure during a Switch Operation

xdcr-bad

```
INVOCATION:
(*FACT (VALUE $XDCR $V))


CONTEXT:
(AND (*FACT (TYPE P-XDCR $XDCR))
     (*FACT (ALTERNATE $XDCR $XDCRA))
     (*FACT (DISAGREE $V (VALUE-OF $XDCRA)))
     (*FACT (STATUS $XDCR GOOD))
     (*FACT (STATUS $XDCRA GOOD))
     (*FACT (ASSOCIATED-UNIT $XDCR $TK)))


GOAL ACHIEVER?:
T


EFFECTS:
NIL


PROPERTIES:
((SAFETY-HANDLER T))
```

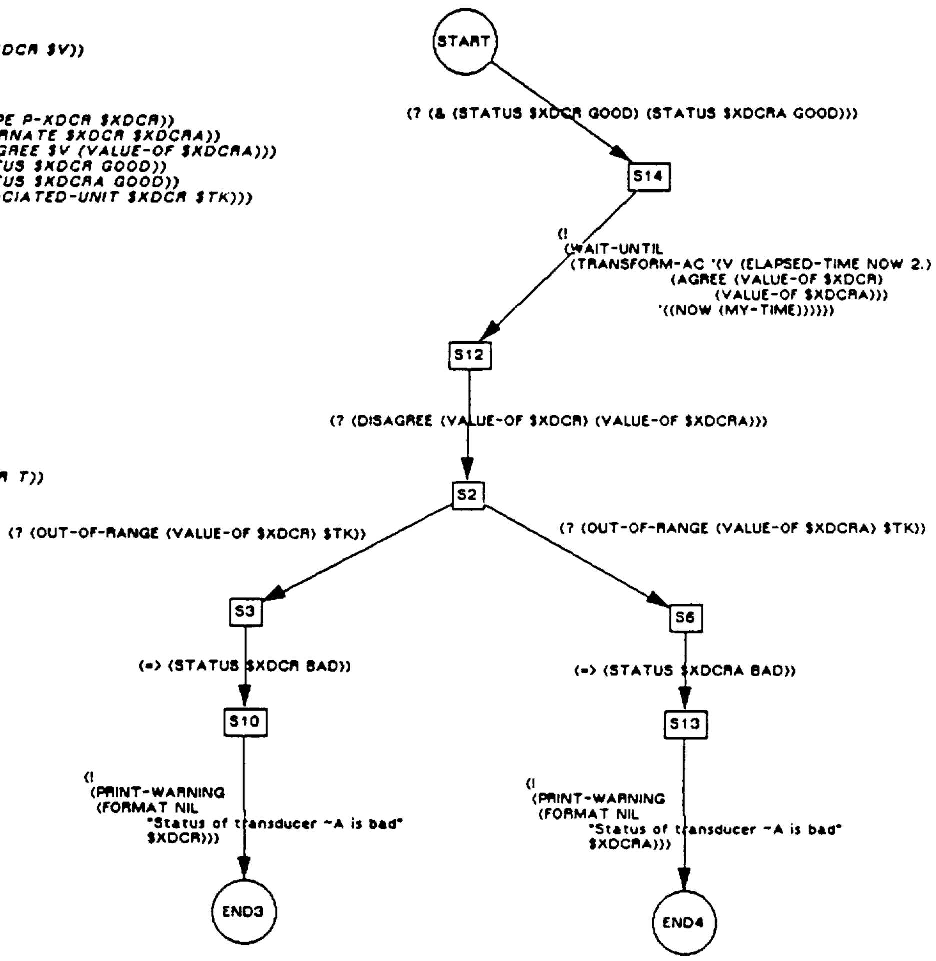

START
(? (& (STATUS $XDCR GOOD) (STATUS $XDCRA GOOD)))
S14
(! (WAIT-UNTIL (TRANSFORM-AC '(V (ELAPSED-TIME NOW 2.) (AGREE (VALUE-OF $XDCR) (VALUE-OF $XDCRA))) '((NOW (MY-TIME))))))
S12
(? (DISAGREE (VALUE-OF $XDCR) (VALUE-OF $XDCRA)))
S2
(? (OUT-OF-RANGE (VALUE-OF $XDCR) $TK))
(? (OUT-OF-RANGE (VALUE-OF $XDCRA) $TK))
S3
S6
(=> (STATUS $XDCR BAD))
(=> (STATUS $XDCRA BAD))
S10
S13
(! (PRINT-WARNING (FORMAT NIL "Status of transducer ~A is bad" $XDCR)))
(! (PRINT-WARNING (FORMAT NIL "Status of transducer ~A is bad" $XDCRA)))
END3
END4

Figure 4.8: The xdcr-bad KA

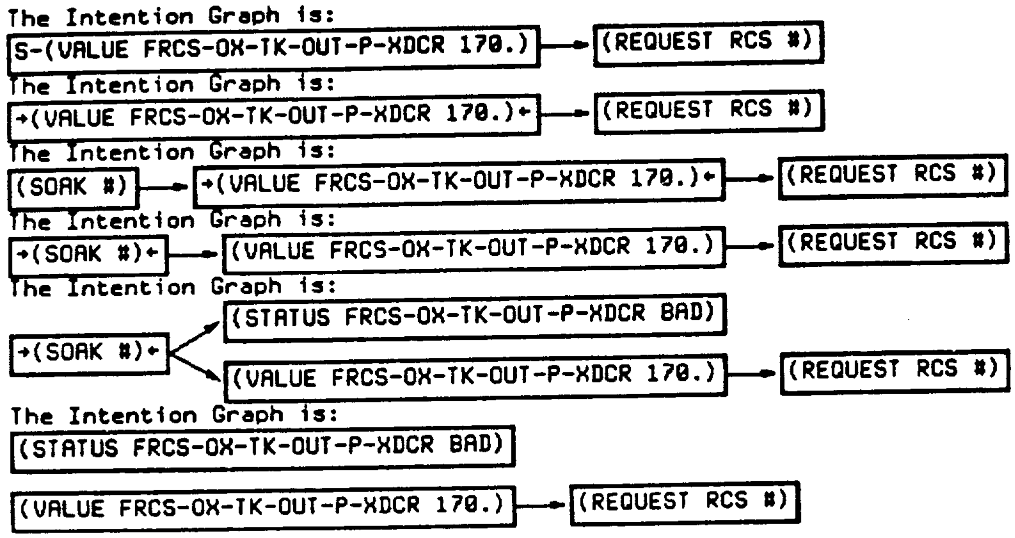

The Intention Graph is:
S-(VALUE FRCS-OX-TK-OUT-P-XDCR 170.)
(REQUEST RCS #)
The Intention Graph is:
→(VALUE FRCS-OX-TK-OUT-P-XDCR 170.)←
(REQUEST RCS #)
The Intention Graph is:
(SOAK #)
→(VALUE FRCS-OX-TK-OUT-P-XDCR 170.)←
(REQUEST RCS #)
The Intention Graph is:
→(SOAK #)←
(VALUE FRCS-OX-TK-OUT-P-XDCR 170.)
(REQUEST RCS #)
The Intention Graph is:
(STATUS FRCS-OX-TK-OUT-P-XDCR BAD)
→(SOAK #)←
(VALUE FRCS-OX-TK-OUT-P-XDCR 170.)
(REQUEST RCS #)
The Intention Graph is:
(STATUS FRCS-OX-TK-OUT-P-XDCR BAD)
(VALUE FRCS-OX-TK-OUT-P-XDCR 170.)
(REQUEST RCS #)

Figure 4.9: Typical Intention Structure for Faulty Transducer

safety handler to all those KAs that should be executed at the earliest possible time. Then we design the metalevel KA that chooses between potentially applicable KAs to place all safety handlers on the intention structure earlier in the ordering than other intentions. In the example given above, the KA that detects the faulty transducer is a safety-handler, and thus is executed prior to servicing the request from RCS. INTERFACE disregards the faulty transducer reading when advising RCS of the pressure, and thus eventually RCS comes to realize that the alarm was activated in error and that the pressure is within normal range.

Even with all this going on, other things are happening within the INTERFACE system. For example, the fact that the transducer is determined to be bad, together with the fact that it is the very transducer that informs the shuttle computers of overpressurization problems, causes the invocation of another KA. In this case, the KA reflects a flight rule that states that overpressurization protection is lost while this transducer is inoperative.

As before, metalevel KAs are invoked to determine which KAs to adopt as intentions and how to order them on the intention structure. The state of the intention structure at one stage during this process is shown in Figure 4.9.

4.2.3 Failed Regulator

It's now time to consider the top-level PRS instantiation, RCS. The case we examine occurs when the regulator on the feed line between the helium tank and its associated propellant tank fails. In this example, we will assume that the `frcs-fu-he-tk-A-reg` has failed. We will focus only on RCS (INTERFACE is, of course, working away during this process as discussed above).

The first thing that happens when the regulator fails is that pressures throughout the fuel subsystem begin to rise. When they exceed the upper limit of 300 psi, certain alarms are activated. This activates the KA `cw-rcs-alarm`, which attempts to confirm that the system is indeed overpressurized.

(Note that this process is more complicated than it first appears. High transducer readings can trigger an alarm which in turn activates the KA `cw-rcs-high` in the PRS system RCS. The high transducer readings that gave rise to the caution-warning alarm will also trigger KAs in the PRS system INTERFACE. These KAs will proceed to verify that the corresponding transducers are not faulty; that is, that the reading of the transducers is indeed accurate. While doing this (or after doing this) INTERFACE will get a request from RCS to advise the latest pressure readings. If INTERFACE is in the process of checking the transducers, it will defer answering this request until it has completed its evaluation of transducer status. But eventually it will return to answering the request and, in the case we are considering, advise that the pressure is indeed above 300 psi.)

On concluding that the system is overpressurized, another KA (`cw-rcs-high`) is activated and this, eventually, concludes that the A regulator has failed (see Figure 4.10). Note that this KA establishes subgoals to close both the A valve and the B valve (there are cases when both are open). For the A valve, this involves a request to INTERFACE as discussed above. However, for the B valve, the system notices that the B valve is already closed. Thus, its goal is directly achieved without the necessity to perform any action or request.

The concluding of this fact then activates another KA (`tk-p-high`) that opens the valve to the alternate (B) regulator. Having opened the valve, it is desirable to then place it under the control of the computer. However, this cannot be done until the pressure in the system drops below 312 psi, as otherwise the computer will automatically shut the valve again. Thus, the malfunction handling procedures specify that the astronaut should wait until this condition is achieved before proceeding to place the valve switch in the GPC position.

RCS achieves this by asking INTERFACE to monitor the pressure and advise it when it drops below 312 psi. While waiting for an answer, the KA `tk-p-high` is suspended (see Figure 4.11).

When the pressure eventually drops below that threshold, the KA (intention) is awakened, and execution continued. Thus, the valve switch is finally placed in the GPC position and the overpressurization problem resolved.[2]

[2]Note the necessity to explicitly remove the fact regarding the response of RCS to the monitoring request. This is necessary with the truth maintenance procedures used by the present system, but clearly needs improvement.

cw-rcs-high

```
INVOCATION:
(*FACT (OVERPRESSURIZED $TK $P-SYS))

  CONTEXT:
  (AND (*FACT (TYPE PROPELLANT-TANK $TK))
       (*FACT (PART-OF $RCS $P-SYS)))
  GOAL ACHIEVER?:
  T

  EFFECTS:
  NIL

  PROPERTIES:
  ((SAFETY-HANDLER T))
```

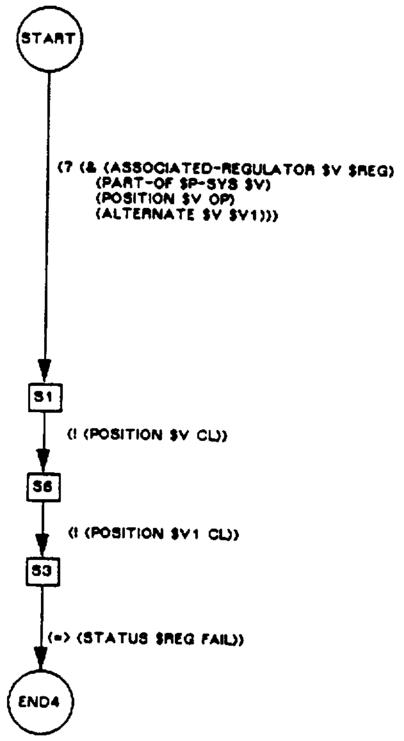


Figure 4.10: The KA cw-rcs-high

tk-p-high

```
INVOCATION:
(AND (*FACT (STATUS $REG FAIL))
     (*FACT (OVERPRESSURIZED $TK $P-SYS)))

CONTEXT:
(AND (*FACT (ALTERNATE $REG $REG1))
     (*FACT (ASSOCIATED-REGULATOR $V1 $REG1))
     (*FACT (SWITCH $V1 $S1)))

GOAL ACHIEVER?:
T

EFFECTS:
(~ (OVERPRESSURIZED $TK $P-SYS))

PROPERTIES:
NIL
```

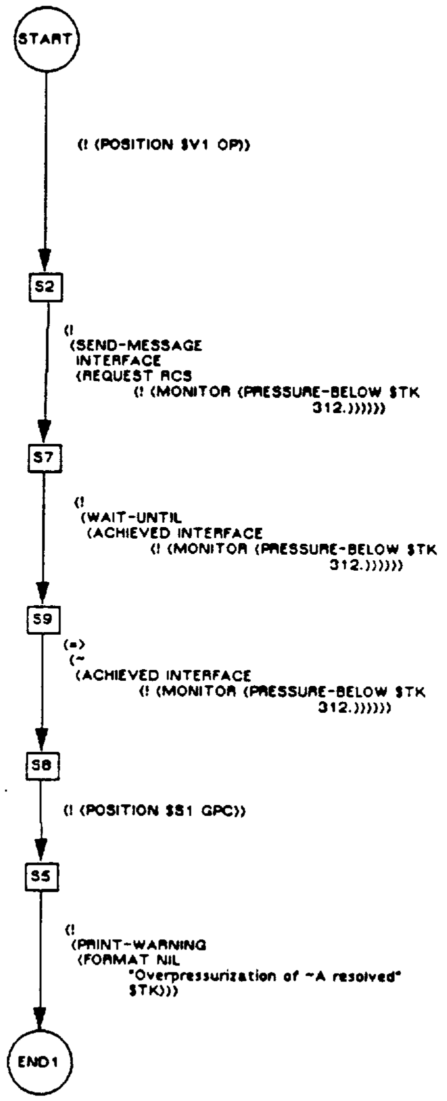

START
(! (POSITION $V1 OP))
S2
(! (SEND-MESSAGE INTERFACE (REQUEST RCS (! (MONITOR (PRESSURE-BELOW $TK 312.))))))
S7
(! (WAIT-UNTIL (ACHIEVED INTERFACE (! (MONITOR (PRESSURE-BELOW $TK 312.))))))
S9
(=) (~ (ACHIEVED INTERFACE (! (MONITOR (PRESSURE-BELOW $TK 312.))))))
S8
(! (POSITION $S1 GPC))
S5
(! (PRINT-WARNING (FORMAT NIL "Overpressurization of ~A resolved" $TK)))
END1

Figure 4.11: The KA **tk-p-high**

4.2.4 Isolating a System Leak

Let's assume that there is a leak in the RCS. Usually, the leak will cause a pressure drop in the system that will set of a caution-warning (cw) alarm. This will cause the KA `cw-rcs-prop-low` to respond. This KA will first try to differentiate between a failed regulator and a leak in the system. If it determines that the system has a leak, it will then establish the goal to isolate that leak. This, in turn, triggers the KA `rcs-leak-isol`. This KA first attempts to secure the system. This involves requesting that the astronauts close all the valves in the leaking system.

(Again, the PRS sytem `INTERACE` will, throughout each process of closing a valve, check that the valve has indeed closed and that the corresponding talkbacks are registering closed. Notice also that the KA used to secure the RCS (`rcs-secure`) includes goals that apply actions to *sets* of objects rather than single ones (see Section 3.4).)

As soon as the system has been secured, the system can identify the leaking section by checking for decreasing pressure in each section of the RCS in turn (using the KA `rcs-leak-isol`).

Chapter 5

Conclusions

The system decribed above was implemented on a Symbolics 3600 LISP machine and has been used to detect and recover from most of the possible malfunctions of the RCS, including sensor faults, leaking components, and regulator and jet failures. This was accomplished by using multiple communicating instantiations of PRS and a simulator for providing real-time input to the system. The experiment provided a severe and positive test of the system's ability to coordinate various plans of action, modify intentions appropriately, and shift its focus of attention. In addition, PRS met every criterion outlined by Laffey et al. [10] for evaluating real-time reasoning systems: high performance, guaranteed response, temporal reasoning capabilities, support for asynchronous inputs, interrupt handling, continuous operation, handling of noisy (possibly inaccurate) data, and shift of focus of attention.

The features of PRS that, we believe, contributed most to its success at this task are (1) its partial planning strategy, (2) its reactivity, (3) its use of procedural knowledge, and (4) its metalevel (reflective) capabilities. At any time, the plans the system is intending to execute (i.e., the selected KAs) are both partial and hierarchical — that is, while certain general goals have been decided upon, the specific means for achieving these ends have been left open for future deliberation. By finding and executing relevant procedures only when needed and only when sufficient information is available for making prudent decisions, the system stands a better chance of achieving its goals under real-time constraints.

The wealth of procedural knowledge possessed by the system is also critical in allowing the system to operate effectively in real-time and to perform a variety of very complex tasks. In particular, the powerful control constructs that can be represented by KAs (such as conditionals and loops) were essential to realizing the malfunction handling procedures used for the shuttle, and allowed low-level system functions to be implemented in the same formalism (i.e.,without resorting to LISP or some other programming language).

PRS also makes it possible to have a large number of diverse KAs available for achieving a goal. Each may vary in its ability to accomplish a goal, as well as in its applicability in particular situations. Thus, if there is insufficient information about a given situation to allow one KA to be used, another (perhaps less reliable) might be available instead. Parallelism and reactivity also help in providing robustness. For example, if one PRS instantiation were busy diagnosing some system fault, other instantiations could remain active, monitoring environmental changes and keeping the spacecraft in a stable and safe configuration.

The metalevel reasoning capabilities of PRS are particularly important in managing the application of the various KAs in different situations. Such capabilities can be critical in deciding how best to meet the real-time constraints of a domain. In particular, the combination of a rich intention structure - supporting multiple active, suspended, and conditional intentions - and the metalevel scheduling capabilities appear to be essential components of complex real-time applications.

Because PRS exhibits behavioral properties similar to those of a rational agent and is ascribed the psychological attitudes of belief, desire, and intention, the potential for a high level of communication with astronauts and other users is greatly enhanced. The astronaut can inquire of the system its current intentions, and determine the reasons for its adopting those intentions. If desired, the astronaut could suggest different means for accomplishing the same task; alternatively, he or she may question the truth of the beliefs upon which the system has based its reasoning and can direct the system to examine the source of these beliefs in greater detail. Of course, the system does not come close to manifesting the behavioral complexity of real rational agents; nevertheless, it represents an important step in building machines that can interact *in a rational way* with human users.

There are many issues that need further study. In particular, it is important to reexamine the semantics of KAs to allow for external events to occur during their execution and for other KAs to interrupt processing. The truth maintenance techniques used for handling the database need further study and extension, and goals of maintenance must be considered. It is necessary to examine more complex scheduling schemes and, finally, to expand the application domain to handle fully and properly the malfunction procedures of the RCS.

Acknowledgments

Amy Lansky and Felix Ingrand assisted greatly in the design of PRS and in working out the theoretical foundations of the system. Pierre Bessiere, Lorna Shinkle, Joshua Singer, and Mabry Tyson helped in the development of PRS and extended the implementation as needed. Dave Wilkins built the simulator and provided valuable advice. The project has also benefited greatly from our participation in and interactions with members of the Rational Agency Group (RATAG) at the Center for the Study of Language and Information, especially Michael Bratman, Phil Cohen, David Israel, Martha Pollack, and others mentioned above.

Bibliography

[1] J. S. Aikins. Prototypical knowledge for expert systems. *Artificial Intelligence*, 20:163–210, 1983.

[2] M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computation Linguistics*, 1988.

[3] R. A. Brooks. A robust layered control system for a mobile robot. Technical Report 864, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1985.

[4] T. Dean and M. Boddy. An analysis of time-dependent planning. Technical report, Department of Computer Science, Brown University, Providence, Rhode Island, 1988.

[5] M. P. Georgeff. An embedded real-time reasoning system. In *Proceedings of the 12th IMACS World Congress*, Paris, France, 1988.

[6] M. P. Georgeff and A. L. Lansky. Procedural knowledge. *Proceedings of the IEEE Special Issue on Knowledge Representation*, 74:1383–1398, 1986.

[7] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning: An experiment with a mobile robot. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, Washington, 1986.

[8] M. P. Georgeff and A. L. Lansky. A system for reasoning in dynamic domains: Fault diagnosis on the space shuttle. Technical Note 375, Artificial Intelligence Center, SRI International, Menlo Park, California, 1986.

[9] L. P. Kaelbling. An architecture for intelligent reactive systems. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 395–410. Morgan Kaufmann, Los Altos, California, 1987.

[10] T. J. Laffey, P. A. Cox, J. L. Schmidt, S. M. Kao, and J. Y. Read. Real-time knowledge-based systems. *AI Magazine*, 9(1):27–45, 1988.

[11] J. Marsh and J. Greenwood. Real-time AI: Software architecture issues. In *Proceedings of the IEEE National Aerospace and Electronics Conference*, pages 67–77, Washington, D.C., 1986.

[12] C. A. O'Reilly and A. S. Cromarty. "Fast" is not "real-time" in designing effective real-time AI systems. In *Applications of Artificial Intelligence II*, pages 249–257, Bellingham, Washington, 1985. Int. Soc. of Optical Engineering.

[13] D. J. Rosencrantz and R. E. Stearns. *Compiler Design.* Academic Press, New York, N.Y., 1978.

[14] D. E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann, Los Altos, California, 1988.

Appendix A

The RCS Database

Electrical Buses

```
(type electrical-bus mma-bus)
(type electrical-bus mmb-bus)
(type electrical-bus mmc-bus)
(type electrical-bus fpc1-bus)
(type electrical-bus ac1-bus)
(type electrical-bus ac2-bus)
(type electrical-bus ac3-bus)
(type electrical-bus ab1-bus)
(type electrical-bus ab2-bus)
(type electrical-bus ab3-bus)
(type electrical-bus bc1-bus)
(type electrical-bus bc2-bus)
(type electrical-bus bc3-bus)
(type electrical-bus ca1-bus)
(type electrical-bus ca2-bus)
(type electrical-bus ca3-bus)
(type electrical-bus flc1-bus)
(type electrical-bus fmc1-bus)
(type electrical-bus fpc2-bus)
(type electrical-bus flc2-bus)
(type electrical-bus fmc2-bus)
(type electrical-bus fpc3-bus)
(type electrical-bus flc3-bus)
(type electrical-bus fmc3-bus)

(negation-as-failure sub-bus)
```

```
(sub-bus mna-bus fpc1-bus)
(sub-bus fpc1-bus flc1-bus)
(sub-bus fpc1-bus fmc1-bus)
(sub-bus fpc1-bus ac1-bus)
(sub-bus mnb-bus fpc2-bus)
(sub-bus fpc2-bus flc2-bus)
(sub-bus fpc2-bus fmc2-bus)
(sub-bus fpc2-bus ac2-bus)
(sub-bus mnc-bus fpc3-bus)
(sub-bus fpc3-bus flc3-bus)
(sub-bus fpc3-bus fmc3-bus)
(sub-bus fpc3-bus ac3-bus)

(functional status 1)

(status mna-bus good)
(status mnb-bus good)
(status mnc-bus good)
(status ac1-bus good)
(status ac2-bus good)
(status ac3-bus good)
(status ab1-bus good)
(status ab2-bus good)
(status ab3-bus good)
(status bc1-bus good)
(status bc2-bus good)
(status bc3-bus good)
(status ca1-bus good)
(status ca2-bus good)
(status ca3-bus good)
(status fpc1-bus good)
(status flc1-bus good)
(status fmc1-bus good)
(status fpc2-bus good)
(status flc2-bus good)
(status fmc2-bus good)
(status fpc3-bus good)
(status flc3-bus good)
(status fmc3-bus good)
```

Computer Buses

```
(type data-path mdm-ff1)
(type data-path mdm-ff2)
(type data-path mdm-ff3)
(type data-path mdm-ff4)

(status mdm-ff1 good)
(status mdm-ff2 good)
(status mdm-ff3 good)
(status mdm-ff4 good)
```

RCS Systems

```
(type rcs frcs)
(type propellant-system frcs-fu)
(type propellant-system frcs-ox)
```

Oxidizer and Fuel Tank Subsystems

```
(type tank frcs-ox-tk)
(type tank frcs-fu-tk)

(type propellant-tank frcs-ox-tk)
(type propellant-tank frcs-fu-tk)

(status frcs-ox-tk good)
(status frcs-fu-tk good)

(type valve frcs-ox-tk-isol-12-valve)
(type valve frcs-ox-tk-isol-345-valve)
(type valve frcs-fu-tk-isol-12-valve)
(type valve frcs-fu-tk-isol-345-valve)

(type propellant-tank-valve frcs-ox-tk-isol-12-valve)
(type propellant-tank-valve frcs-ox-tk-isol-345-valve)
```

```
(type propellant-tank-valve frcs-fu-tk-isol-12-valve)
(type propellant-tank-valve frcs-fu-tk-isol-345-valve)

(status frcs-ox-tk-isol-12-valve good)
(status frcs-ox-tk-isol-345-valve good)
(status frcs-fu-tk-isol-12-valve good)
(status frcs-fu-tk-isol-345-valve good)
```

following values are op, cl

```
(functional position 1)
(negation-as-failure position)

(position frcs-ox-tk-isol-12-valve op)
(position frcs-ox-tk-isol-345-valve op)
(position frcs-fu-tk-isol-12-valve op)
(position frcs-fu-tk-isol-345-valve op)

(type switch frcs-prop-tk-isol-12-switch)
(type switch frcs-prop-tk-isol-345-switch)

(switch frcs-ox-tk-isol-12-valve frcs-prop-tk-isol-12-switch)
(switch frcs-ox-tk-isol-345-valve frcs-prop-tk-isol-345-switch)
(switch frcs-fu-tk-isol-12-valve frcs-prop-tk-isol-12-switch)
(switch frcs-fu-tk-isol-345-valve frcs-prop-tk-isol-345-switch)

(talkback frcs-ox-tk-isol-12-valve frcs-prop-tk-isol-12-talkback)
(talkback frcs-ox-tk-isol-345-valve frcs-prop-tk-isol-345-talkback)
(talkback frcs-fu-tk-isol-12-valve frcs-prop-tk-isol-12-talkback)
(talkback frcs-fu-tk-isol-345-valve frcs-prop-tk-isol-345-talkback)

(associated-talkback frcs-prop-tk-isol-12-switch frcs-prop-tk-isol-12-talkback)
(associated-talkback frcs-prop-tk-isol-345-switch frcs-prop-tk-isol-345-talkback)
```

following values are man-op, man-cl, gpc-op, gpc-cl note drive-capability fact-invoked on status

```
(negation-as-failure drive-capability)

(drive-capability frcs-ox-tk-isol-12-valve man-op)
(drive-capability frcs-ox-tk-isol-12-valve man-cl)
(drive-capability frcs-ox-tk-isol-12-valve gpc-op)
(drive-capability frcs-ox-tk-isol-12-valve gpc-cl)
(drive-capability frcs-ox-tk-isol-345-valve man-op)
(drive-capability frcs-ox-tk-isol-345-valve man-cl)
(drive-capability frcs-ox-tk-isol-345-valve gpc-op)
(drive-capability frcs-ox-tk-isol-345-valve gpc-cl)
(drive-capability frcs-fu-tk-isol-12-valve man-op)
(drive-capability frcs-fu-tk-isol-12-valve man-cl)
(drive-capability frcs-fu-tk-isol-12-valve gpc-op)
(drive-capability frcs-fu-tk-isol-12-valve gpc-cl)
(drive-capability frcs-fu-tk-isol-345-valve man-op)
(drive-capability frcs-fu-tk-isol-345-valve man-cl)
(drive-capability frcs-fu-tk-isol-345-valve gpc-op)
(drive-capability frcs-fu-tk-isol-345-valve gpc-cl)

(type p-xdcr frcs-ox-tk-p-xdcr)
(type p-xdcr frcs-ox-tk-out-p-xdcr)
(type p-xdcr frcs-fu-tk-p-xdcr)
(type p-xdcr frcs-fu-tk-out-p-xdcr)

(status frcs-ox-tk-p-xdcr good)
(status frcs-ox-tk-out-p-xdcr good)
(status frcs-fu-tk-p-xdcr good)
(status frcs-fu-tk-out-p-xdcr good)

(pressure-ll frcs-ox-tk 230)
(pressure-ll frcs-fu-tk 230)
(pressure-ul frcs-ox-tk 290)
(pressure-ul frcs-fu-tk 290)

(functional value 1)

(value frcs-ox-tk-p-xdcr 245)
(value frcs-ox-tk-out-p-xdcr 245)
(value frcs-fu-tk-p-xdcr 245)
(value frcs-fu-tk-out-p-xdcr 245)
```

```
(functional pressure 1)

(pressure frcs-ox-tk 245)
(pressure frcs-fu-tk 245)
```

Helium Tank Subsystems

```
(type tank frcs-ox-he-tk)
(type tank frcs-fu-he-tk)

(type he-tank frcs-ox-he-tk)
(type he-tank frcs-fu-he-tk)

(type valve frcs-ox-he-tk-isol-A-valve)
(type valve frcs-ox-he-tk-isol-B-valve)
(type valve frcs-fu-he-tk-isol-A-valve)
(type valve frcs-fu-he-tk-isol-B-valve)

(type he-tank-valve frcs-ox-he-tk-isol-A-valve)
(type he-tank-valve frcs-ox-he-tk-isol-B-valve)
(type he-tank-valve frcs-fu-he-tk-isol-A-valve)
(type he-tank-valve frcs-fu-he-tk-isol-B-valve)

(status frcs-ox-he-tk-isol-A-valve good)
(status frcs-ox-he-tk-isol-B-valve good)
(status frcs-fu-he-tk-isol-A-valve good)
(status frcs-fu-he-tk-isol-B-valve good)
```

following values are op, cl

```
(position frcs-ox-he-tk-isol-A-valve op)
(position frcs-ox-he-tk-isol-B-valve cl)
(position frcs-fu-he-tk-isol-A-valve op)
(position frcs-fu-he-tk-isol-B-valve cl)

(type switch frcs-he-tk-isol-A-switch)
(type switch frcs-he-tk-isol-B-switch)

(switch frcs-ox-he-tk-isol-A-valve frcs-he-tk-isol-A-switch)
```

```
(switch frcs-ox-he-tk-isol-B-valve frcs-he-tk-isol-B-switch)
(switch frcs-fu-he-tk-isol-A-valve frcs-he-tk-isol-A-switch)
(switch frcs-fu-he-tk-isol-B-valve frcs-he-tk-isol-B-switch)

(talkback frcs-ox-he-tk-isol-A-valve frcs-he-tk-isol-A-talkback)
(talkback frcs-ox-he-tk-isol-B-valve frcs-he-tk-isol-B-talkback)
(talkback frcs-fu-he-tk-isol-A-valve frcs-he-tk-isol-A-talkback)
(talkback frcs-fu-he-tk-isol-B-valve frcs-he-tk-isol-B-talkback)

(associated-talkback frcs-he-tk-isol-A-switch frcs-he-tk-isol-A-talkback)
(associated-talkback frcs-he-tk-isol-B-switch frcs-he-tk-isol-B-talkback)
```

following values are man-op, man-cl, gpc-op, gpc-cl

```
(drive-capability frcs-ox-he-tk-isol-A-valve man-op)
(drive-capability frcs-ox-he-tk-isol-A-valve man-cl)
(drive-capability frcs-ox-he-tk-isol-A-valve gpc-op)
(drive-capability frcs-ox-he-tk-isol-A-valve gpc-cl)
(drive-capability frcs-ox-he-tk-isol-B-valve man-op)
(drive-capability frcs-ox-he-tk-isol-B-valve man-cl)
(drive-capability frcs-ox-he-tk-isol-B-valve gpc-op)
(drive-capability frcs-ox-he-tk-isol-B-valve gpc-cl)
(drive-capability frcs-fu-he-tk-isol-A-valve man-op)
(drive-capability frcs-fu-he-tk-isol-A-valve man-cl)
(drive-capability frcs-fu-he-tk-isol-A-valve gpc-op)
(drive-capability frcs-fu-he-tk-isol-A-valve gpc-cl)
(drive-capability frcs-fu-he-tk-isol-B-valve man-op)
(drive-capability frcs-fu-he-tk-isol-B-valve man-cl)
(drive-capability frcs-fu-he-tk-isol-B-valve gpc-op)
(drive-capability frcs-fu-he-tk-isol-B-valve gpc-cl)

(type p-xdcr frcs-ox-he-tk-p-1-xdcr)
(type p-xdcr frcs-ox-he-tk-p-2-xdcr)
(type p-xdcr frcs-fu-he-tk-p-1-xdcr)
(type p-xdcr frcs-fu-he-tk-p-2-xdcr)

(status frcs-ox-he-tk-p-1-xdcr good)
(status frcs-ox-he-tk-p-2-xdcr good)
(status frcs-fu-he-tk-p-1-xdcr good)
(status frcs-fu-he-tk-p-2-xdcr good)
```

```
(pressure-ll frcs-ox-he-tk 500)
(pressure-ll frcs-fu-he-tk 500)
(pressure-ul frcs-ox-he-tk 50000)
(pressure-ul frcs-fu-he-tk 50000)

(value frcs-ox-he-tk-p-1-xdcr 3000)
(value frcs-ox-he-tk-p-2-xdcr 3000)
(value frcs-fu-he-tk-p-1-xdcr 3000)
(value frcs-fu-he-tk-p-2-xdcr 3000)

(pressure frcs-ox-he-tk 3000)
(pressure frcs-fu-he-tk 3000)

(type regulator frcs-ox-he-tk-A-reg)
(type regulator frcs-ox-he-tk-B-reg)
(type regulator frcs-fu-he-tk-A-reg)
(type regulator frcs-fu-he-tk-B-reg)

(status frcs-ox-he-tk-A-reg good)
(status frcs-ox-he-tk-B-reg good)
(status frcs-fu-he-tk-A-reg good)
(status frcs-fu-he-tk-B-reg good)
```

Manifolds

```
(type manifold frcs-ox-manf-1)
(type manifold frcs-ox-manf-2)
(type manifold frcs-ox-manf-3)
(type manifold frcs-ox-manf-4)
(type manifold frcs-ox-manf-5)
(type manifold frcs-fu-manf-1)
(type manifold frcs-fu-manf-2)
(type manifold frcs-fu-manf-3)
(type manifold frcs-fu-manf-4)
(type manifold frcs-fu-manf-5)

(type valve frcs-ox-manf-1-valve)
(type valve frcs-ox-manf-2-valve)
(type valve frcs-ox-manf-3-valve)
```

```
(type valve frcs-ox-manf-4-valve)
(type valve frcs-ox-manf-5-valve)
(type valve frcs-fu-manf-1-valve)
(type valve frcs-fu-manf-2-valve)
(type valve frcs-fu-manf-3-valve)
(type valve frcs-fu-manf-4-valve)
(type valve frcs-fu-manf-5-valve)

(type manifold-valve frcs-ox-manf-1-valve)
(type manifold-valve frcs-ox-manf-2-valve)
(type manifold-valve frcs-ox-manf-3-valve)
(type manifold-valve frcs-ox-manf-4-valve)
(type manifold-valve frcs-ox-manf-5-valve)
(type manifold-valve frcs-fu-manf-1-valve)
(type manifold-valve frcs-fu-manf-2-valve)
(type manifold-valve frcs-fu-manf-3-valve)
(type manifold-valve frcs-fu-manf-4-valve)
(type manifold-valve frcs-fu-manf-5-valve)

(status frcs-ox-manf-1-valve good)
(status frcs-ox-manf-2-valve good)
(status frcs-ox-manf-3-valve good)
(status frcs-ox-manf-4-valve good)
(status frcs-ox-manf-5-valve good)
(status frcs-fu-manf-1-valve good)
(status frcs-fu-manf-2-valve good)
(status frcs-fu-manf-3-valve good)
(status frcs-fu-manf-4-valve good)
(status frcs-fu-manf-5-valve good)

(position frcs-ox-manf-1-valve op)
(position frcs-ox-manf-2-valve op)
(position frcs-ox-manf-3-valve op)
(position frcs-ox-manf-4-valve op)
(position frcs-ox-manf-5-valve op)
(position frcs-fu-manf-1-valve op)
(position frcs-fu-manf-2-valve op)
(position frcs-fu-manf-3-valve op)
(position frcs-fu-manf-4-valve op)
(position frcs-fu-manf-5-valve op)
```

```
(type switch frcs-manf-1-switch)
(type switch frcs-manf-2-switch)
(type switch frcs-manf-3-switch)
(type switch frcs-manf-4-switch)
(type switch frcs-manf-5-switch)

(switch frcs-ox-manf-1-valve frcs-manf-1-switch)
(switch frcs-ox-manf-2-valve frcs-manf-2-switch)
(switch frcs-ox-manf-3-valve frcs-manf-3-switch)
(switch frcs-ox-manf-4-valve frcs-manf-4-switch)
(switch frcs-ox-manf-5-valve frcs-manf-5-switch)
(switch frcs-fu-manf-1-valve frcs-manf-1-switch)
(switch frcs-fu-manf-2-valve frcs-manf-2-switch)
(switch frcs-fu-manf-3-valve frcs-manf-3-switch)
(switch frcs-fu-manf-4-valve frcs-manf-4-switch)
(switch frcs-fu-manf-5-valve frcs-manf-5-switch)

(associated-talkback frcs-manf-1-switch frcs-manf-1-talkback)
(associated-talkback frcs-manf-2-switch frcs-manf-2-talkback)
(associated-talkback frcs-manf-3-switch frcs-manf-3-talkback)
(associated-talkback frcs-manf-4-switch frcs-manf-4-talkback)
(associated-talkback frcs-manf-5-switch frcs-manf-5-talkback)

(talkback frcs-ox-manf-1-valve frcs-manf-1-talkback)
(talkback frcs-ox-manf-2-valve frcs-manf-2-talkback)
(talkback frcs-ox-manf-3-valve frcs-manf-3-talkback)
(talkback frcs-ox-manf-4-valve frcs-manf-4-talkback)
(talkback frcs-ox-manf-5-valve frcs-manf-5-talkback)
(talkback frcs-fu-manf-1-valve frcs-manf-1-talkback)
(talkback frcs-fu-manf-2-valve frcs-manf-2-talkback)
(talkback frcs-fu-manf-3-valve frcs-manf-3-talkback)
(talkback frcs-fu-manf-4-valve frcs-manf-4-talkback)
(talkback frcs-fu-manf-5-valve frcs-manf-5-talkback)

(drive-capability frcs-ox-manf-1-valve man-op)
(drive-capability frcs-ox-manf-2-valve man-op)
(drive-capability frcs-ox-manf-3-valve man-op)
(drive-capability frcs-ox-manf-4-valve man-op)
(drive-capability frcs-ox-manf-5-valve man-op)
```

```
(drive-capability frcs-fu-manf-1-valve man-op)
(drive-capability frcs-fu-manf-2-valve man-op)
(drive-capability frcs-fu-manf-3-valve man-op)
(drive-capability frcs-fu-manf-4-valve man-op)
(drive-capability frcs-fu-manf-5-valve man-op)
(drive-capability frcs-ox-manf-1-valve man-cl)
(drive-capability frcs-ox-manf-2-valve man-cl)
(drive-capability frcs-ox-manf-3-valve man-cl)
(drive-capability frcs-ox-manf-4-valve man-cl)
(drive-capability frcs-ox-manf-5-valve man-cl)
(drive-capability frcs-fu-manf-1-valve man-cl)
(drive-capability frcs-fu-manf-2-valve man-cl)
(drive-capability frcs-fu-manf-3-valve man-cl)
(drive-capability frcs-fu-manf-4-valve man-cl)
(drive-capability frcs-fu-manf-5-valve man-cl)
(drive-capability frcs-ox-manf-1-valve gpc-cl)
(drive-capability frcs-ox-manf-2-valve gpc-cl)
(drive-capability frcs-ox-manf-3-valve gpc-cl)
(drive-capability frcs-ox-manf-4-valve gpc-cl)
(drive-capability frcs-ox-manf-5-valve gpc-cl)
(drive-capability frcs-fu-manf-1-valve gpc-cl)
(drive-capability frcs-fu-manf-2-valve gpc-cl)
(drive-capability frcs-fu-manf-3-valve gpc-cl)
(drive-capability frcs-fu-manf-4-valve gpc-cl)
(drive-capability frcs-fu-manf-5-valve gpc-cl)
(drive-capability frcs-ox-manf-1-valve gpc-op)
(drive-capability frcs-ox-manf-2-valve gpc-op)
(drive-capability frcs-ox-manf-3-valve gpc-op)
(drive-capability frcs-ox-manf-4-valve gpc-op)
(drive-capability frcs-ox-manf-5-valve gpc-op)
(drive-capability frcs-fu-manf-1-valve gpc-op)
(drive-capability frcs-fu-manf-2-valve gpc-op)
(drive-capability frcs-fu-manf-3-valve gpc-op)
(drive-capability frcs-fu-manf-4-valve gpc-op)
(drive-capability frcs-fu-manf-5-valve gpc-op)

(pressure frcs-ox-manf-1 245)
(pressure frcs-ox-manf-2 245)
(pressure frcs-ox-manf-3 245)
(pressure frcs-ox-manf-4 245)
```

```
(pressure frcs-fu-manf-1 245)
(pressure frcs-fu-manf-2 245)
(pressure frcs-fu-manf-3 245)
(pressure frcs-fu-manf-4 245)

(type p-xdcr frcs-ox-manf-1-p-xdcr)
(type p-xdcr frcs-ox-manf-2-p-xdcr)
(type p-xdcr frcs-ox-manf-3-p-xdcr)
(type p-xdcr frcs-ox-manf-4-p-xdcr)
(type p-xdcr frcs-fu-manf-1-p-xdcr)
(type p-xdcr frcs-fu-manf-2-p-xdcr)
(type p-xdcr frcs-fu-manf-3-p-xdcr)
(type p-xdcr frcs-fu-manf-4-p-xdcr)

(status frcs-ox-manf-1-p-xdcr good)
(status frcs-ox-manf-2-p-xdcr good)
(status frcs-ox-manf-3-p-xdcr good)
(status frcs-ox-manf-4-p-xdcr good)
(status frcs-fu-manf-1-p-xdcr good)
(status frcs-fu-manf-2-p-xdcr good)
(status frcs-fu-manf-3-p-xdcr good)
(status frcs-fu-manf-4-p-xdcr good)

(value frcs-ox-manf-1-p-xdcr 245)
(value frcs-ox-manf-2-p-xdcr 245)
(value frcs-ox-manf-3-p-xdcr 245)
(value frcs-ox-manf-4-p-xdcr 245)
(value frcs-fu-manf-1-p-xdcr 245)
(value frcs-fu-manf-2-p-xdcr 245)
(value frcs-fu-manf-3-p-xdcr 245)
(value frcs-fu-manf-4-p-xdcr 245)

(status frcs-ox-manf-1 good)
(status frcs-ox-manf-2 good)
(status frcs-ox-manf-3 good)
(status frcs-ox-manf-4 good)
(status frcs-ox-manf-5 good)
(status frcs-fu-manf-1 good)
(status frcs-fu-manf-2 good)
(status frcs-fu-manf-3 good)
```

```
(status frcs-fu-manf-4 good)
(status frcs-fu-manf-5 good)

(manifold-1 frcs-ox frcs-ox-manf-1)
(manifold-2 frcs-ox frcs-ox-manf-2)
(manifold-3 frcs-ox frcs-ox-manf-3)
(manifold-4 frcs-ox frcs-ox-manf-4)
(manifold-5 frcs-ox frcs-ox-manf-5)
(manifold-1 frcs-fu frcs-fu-manf-1)
(manifold-2 frcs-fu frcs-fu-manf-2)
(manifold-3 frcs-fu frcs-fu-manf-3)
(manifold-4 frcs-fu frcs-fu-manf-4)
(manifold-5 frcs-fu frcs-fu-manf-5)
```

Legs

```
(type leg frcs-fu-tk-12-leg)
(type leg frcs-ox-tk-12-leg)
(type leg frcs-fu-tk-345-leg)
(type leg frcs-ox-tk-345-leg)
(type leg frcs-fu-he-tk-A-leg)
(type leg frcs-ox-he-tk-A-leg)
(type leg frcs-fu-he-tk-B-leg)
(type leg frcs-ox-he-tk-B-leg)

(type valve frcs-fu-tk-quad-check-valve)
(type valve frcs-ox-tk-quad-check-valve)

(type quad-check-valve frcs-fu-tk-quad-check-valve)
(type quad-check-valve frcs-ox-tk-quad-check-valve)
```

Miscellaneous

```
(functional quantity 1)

(quantity frcs-ox-he-tk 40)
(quantity frcs-fu-he-tk 40)

(quantity frcs-ox-tk 40)
(quantity frcs-fu-tk 40)
```

```
(alarm-initiator frcs-fu-tk-out-p-xdcr)
(alarm-initiator frcs-ox-tk-out-p-xdcr)
```

System Structure

```
(part-of frcs frcs-fu)
(part-of frcs frcs-ox)

(part-of frcs-ox frcs-ox-he-tk)
(part-of frcs-fu frcs-fu-he-tk)
(part-of frcs-ox frcs-ox-tk)
(part-of frcs-fu frcs-fu-tk)

(part-of frcs-fu frcs-fu-he-tk-A-reg)
(part-of frcs-ox frcs-ox-he-tk-A-reg)
(part-of frcs-fu frcs-fu-he-tk-B-reg)
(part-of frcs-ox frcs-ox-he-tk-B-reg)

(part-of frcs-ox frcs-ox-manf-1)
(part-of frcs-ox frcs-ox-manf-2)
(part-of frcs-ox frcs-ox-manf-3)
(part-of frcs-ox frcs-ox-manf-4)
(part-of frcs-ox frcs-ox-manf-5)
(part-of frcs-fu frcs-fu-manf-1)
(part-of frcs-fu frcs-fu-manf-2)
(part-of frcs-fu frcs-fu-manf-3)
(part-of frcs-fu frcs-fu-manf-4)
(part-of frcs-fu frcs-fu-manf-5)

(part-of frcs-ox frcs-ox-tk-isol-12-valve)
(part-of frcs-ox frcs-ox-tk-isol-345-valve)
(part-of frcs-fu frcs-fu-tk-isol-12-valve)
(part-of frcs-fu frcs-fu-tk-isol-345-valve)

(part-of frcs-ox frcs-ox-he-tk-isol-A-valve)
(part-of frcs-ox frcs-ox-he-tk-isol-B-valve)
(part-of frcs-fu frcs-fu-he-tk-isol-A-valve)
(part-of frcs-fu frcs-fu-he-tk-isol-B-valve)
```

```
(part-of frcs-fu frcs-fu-tk-quad-check-valve)
(part-of frcs-ox frcs-ox-tk-quad-check-valve)

(part-of frcs-ox frcs-ox-manf-1-valve)
(part-of frcs-ox frcs-ox-manf-2-valve)
(part-of frcs-ox frcs-ox-manf-3-valve)
(part-of frcs-ox frcs-ox-manf-4-valve)
(part-of frcs-ox frcs-ox-manf-5-valve)
(part-of frcs-fu frcs-fu-manf-1-valve)
(part-of frcs-fu frcs-fu-manf-2-valve)
(part-of frcs-fu frcs-fu-manf-3-valve)
(part-of frcs-fu frcs-fu-manf-4-valve)
(part-of frcs-fu frcs-fu-manf-5-valve)

(part-of frcs-fu frcs-fu-tk-12-leg)
(part-of frcs-ox frcs-ox-tk-12-leg)
(part-of frcs-fu frcs-fu-tk-345-leg)
(part-of frcs-ox frcs-ox-tk-345-leg)
(part-of frcs-fu frcs-fu-he-tk-A-leg)
(part-of frcs-ox frcs-ox-he-tk-A-leg)
(part-of frcs-fu frcs-fu-he-tk-B-leg)
(part-of frcs-ox frcs-ox-he-tk-B-leg)

(part-of frcs-fu frcs-fu-tk-quad-check-valve)
(part-of frcs-ox frcs-ox-tk-quad-check-valve)

(associated-unit frcs-ox-tk-p-xdcr frcs-ox-tk)
(associated-unit frcs-fu-tk-p-xdcr frcs-fu-tk)
(associated-unit frcs-ox-tk-out-p-xdcr frcs-ox-tk)
(associated-unit frcs-fu-tk-out-p-xdcr frcs-fu-tk)
(associated-unit frcs-ox-he-tk-p-2-xdcr frcs-ox-he-tk)
(associated-unit frcs-fu-he-tk-p-2-xdcr frcs-fu-he-tk)
(associated-unit frcs-ox-he-tk-p-1-xdcr frcs-ox-he-tk)
(associated-unit frcs-fu-he-tk-p-1-xdcr frcs-fu-he-tk)

(associated-unit frcs-ox-manf-1-p-xdcr frcs-ox-manf-1)
(associated-unit frcs-ox-manf-2-p-xdcr frcs-ox-manf-2)
(associated-unit frcs-ox-manf-3-p-xdcr frcs-ox-manf-3)
(associated-unit frcs-ox-manf-4-p-xdcr frcs-ox-manf-4)
(associated-unit frcs-fu-manf-1-p-xdcr frcs-fu-manf-1)
```

```
(associated-unit frcs-fu-manf-2-p-xdcr frcs-fu-manf-2)
(associated-unit frcs-fu-manf-3-p-xdcr frcs-fu-manf-3)
(associated-unit frcs-fu-manf-4-p-xdcr frcs-fu-manf-4)

(connects frcs-ox-he-tk-isol-A-valve frcs-ox-he-tk frcs-ox-he-tk-leg)
(connects frcs-ox-he-tk-isol-B-valve frcs-ox-he-tk frcs-ox-he-tk-leg)
(connects frcs-fu-he-tk-isol-A-valve frcs-fu-he-tk frcs-fu-he-tk-leg)
(connects frcs-fu-he-tk-isol-B-valve frcs-fu-he-tk frcs-fu-he-tk-leg)

(connects frcs-fu-tk-quad-check-valve frcs-fu-he-tk-leg frcs-fu-tk)
(connects frcs-ox-tk-quad-check-valve frcs-ox-he-tk-leg frcs-ox-tk)

(connects frcs-ox-tk-isol-12-valve frcs-ox-tk frcs-ox-tk-12-leg)
(connects frcs-ox-tk-isol-345-valve frcs-ox-tk frcs-ox-tk-345-leg)
(connects frcs-fu-tk-isol-12-valve frcs-fu-tk frcs-fu-tk-12-leg)
(connects frcs-fu-tk-isol-345-valve frcs-fu-tk frcs-fu-tk-345-leg)

(connects frcs-ox-manf-1-valve frcs-ox-tk-12-leg frcs-ox-manf-1)
(connects frcs-ox-manf-2-valve frcs-ox-tk-12-leg frcs-ox-manf-2)
(connects frcs-ox-manf-3-valve frcs-ox-tk-345-leg frcs-ox-manf-3)
(connects frcs-ox-manf-4-valve frcs-ox-tk-345-leg frcs-ox-manf-4)
(connects frcs-ox-manf-5-valve frcs-ox-tk-345-leg frcs-ox-manf-5)
(connects frcs-fu-manf-1-valve frcs-fu-tk-12-leg frcs-fu-manf-1)
(connects frcs-fu-manf-2-valve frcs-fu-tk-12-leg frcs-fu-manf-2)
(connects frcs-fu-manf-3-valve frcs-fu-tk-345-leg frcs-fu-manf-3)
(connects frcs-fu-manf-4-valve frcs-fu-tk-345-leg frcs-fu-manf-4)
(connects frcs-fu-manf-5-valve frcs-fu-tk-345-leg frcs-fu-manf-5)

(negation-as-failure alternate)

(alternate frcs-fu-he-tk-isol-A-valve frcs-fu-he-tk-isol-B-valve)
(alternate frcs-fu-he-tk-isol-B-valve frcs-fu-he-tk-isol-A-valve)
(alternate frcs-ox-he-tk-isol-A-valve frcs-ox-he-tk-isol-B-valve)
(alternate frcs-ox-he-tk-isol-B-valve frcs-ox-he-tk-isol-A-valve)
(alternate frcs-fu-he-tk-A-reg frcs-fu-he-tk-B-reg)
(alternate frcs-fu-he-tk-B-reg frcs-fu-he-tk-A-reg)
(alternate frcs-ox-he-tk-A-reg frcs-fu-he-tk-B-reg)
(alternate frcs-ox-he-tk-B-reg frcs-fu-he-tk-A-reg)
(alternate frcs-ox-tk-p-xdcr frcs-ox-tk-out-p-xdcr)
(alternate frcs-ox-tk-out-p-xdcr frcs-ox-tk-p-xdcr)
```

```
(alternate frcs-fu-tk-out-p-xdcr frcs-fu-tk-p-xdcr)
(alternate frcs-fu-tk-p-xdcr frcs-fu-tk-out-p-xdcr)
(alternate frcs-ox-he-tk-p-1-xdcr frcs-ox-he-tk-p-2-xdcr)
(alternate frcs-ox-he-tk-p-2-xdcr frcs-ox-he-tk-p-1-xdcr)
(alternate frcs-fu-he-tk-p-2-xdcr frcs-fu-he-tk-p-1-xdcr)
(alternate frcs-fu-he-tk-p-1-xdcr frcs-fu-he-tk-p-2-xdcr)

(associated-regulator frcs-fu-he-tk-isol-A-valve frcs-fu-he-tk-A-reg)
(associated-regulator frcs-fu-he-tk-isol-B-valve frcs-fu-he-tk-b-reg)
(associated-regulator frcs-ox-he-tk-isol-A-valve frcs-ox-he-tk-A-reg)
(associated-regulator frcs-ox-he-tk-isol-B-valve frcs-ox-he-tk-b-reg)

(other-propellant-system frcs-fu frcs-ox)
(other-propellant-system frcs-ox frcs-fu)


(functional mode 2)

(~ (gpc-display frcs-fu tk-p))
(~ (gpc-display frcs-ox tk-p))
```

Appendix B

The RCS Procedures

The various KAs used in the RCS KA library are given on the following pages.

tk-p-high

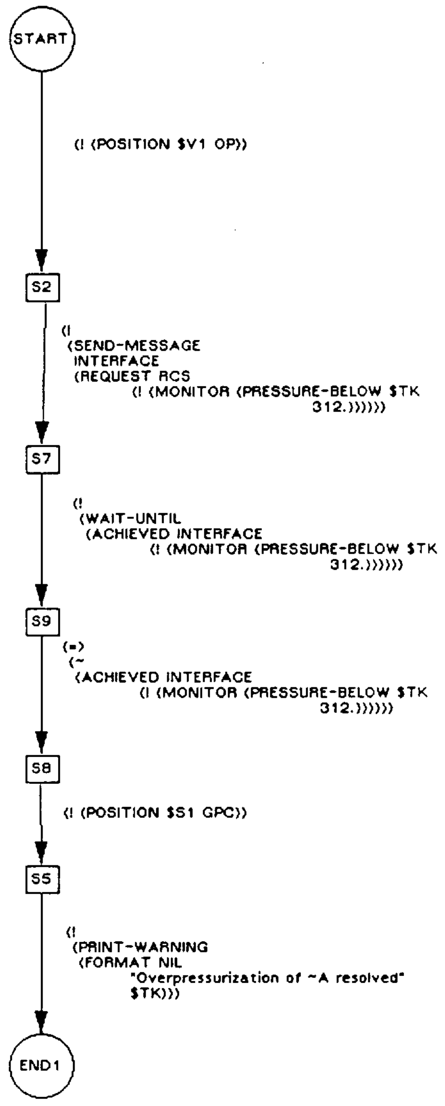
INVOCATION:
```
(AND (*FACT (STATUS $REG FAIL))
     (*FACT (OVERPRESSURIZED $TK $P-SYS)))
```

CONTEXT:
```
(AND (*FACT (ALTERNATE $REG $REG1))
     (*FACT (ASSOCIATED-REGULATOR $V1 $REG1))
     (*FACT (SWITCH $V1 $S1)))
```

GOAL ACHIEVER?:
T

EFFECTS:
```
(~ (OVERPRESSURIZED $TK $P-SYS))
```

PROPERTIES:
NIL



```
(! (POSITION $V1 OP))
```

```
(!
 (SEND-MESSAGE
  INTERFACE
  (REQUEST RCS
          (! (MONITOR (PRESSURE-BELOW $TK
                                        312.))))))
```

```
(!
 (WAIT-UNTIL
  (ACHIEVED INTERFACE
           (! (MONITOR (PRESSURE-BELOW $TK
                                         312.))))))
```

```
(=)
(~
 (ACHIEVED INTERFACE
          (! (MONITOR (PRESSURE-BELOW $TK
                                        312.))))))
```

```
(! (POSITION $S1 GPC))
```

```
(!
 (PRINT-WARNING
  (FORMAT NIL
          "Overpressurization of ~A resolved"
          $TK)))
```

cw-rcs-high

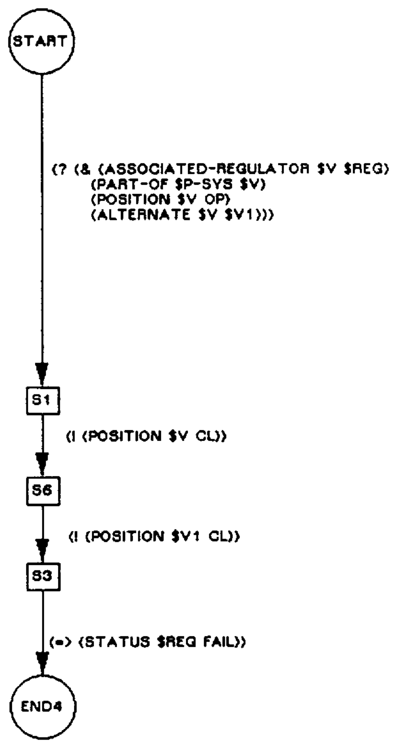
INVOCATION:
*(*FACT (OVERPRESSURIZED $TK $P-SYS))*

CONTEXT:
*(AND (*FACT (TYPE PROPELLANT-TANK $TK))*
*(*FACT (PART-OF $RCS $P-SYS)))*

GOAL ACHIEVER?:
T

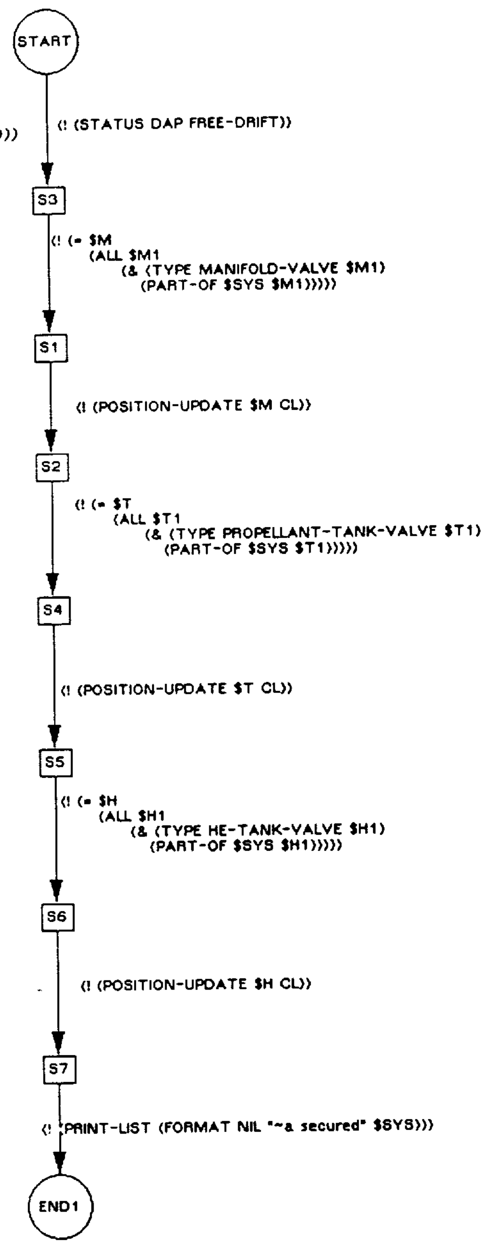
EFFECTS:
NIL

PROPERTIES:
((SAFETY-HANDLER T))



rcs-secure

```
INVOCATION:
(AND (*GOAL (! (SECURED $SYS)))
     (*FACT (TYPE PROPELLANT-SYSTEM $SYS)))

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL
```



```
(! (STATUS DAP FREE-DRIFT))

(! (= $M
      (ALL $M1
           (& (TYPE MANIFOLD-VALVE $M1)
              (PART-OF $SYS $M1)))))

(! (POSITION-UPDATE $M CL))

(! (= $T
      (ALL $T1
           (& (TYPE PROPELLANT-TANK-VALVE $T1)
              (PART-OF $SYS $T1)))))

(! (POSITION-UPDATE $T CL))

(! (= $H
      (ALL $H1
           (& (TYPE HE-TANK-VALVE $H1)
              (PART-OF $SYS $H1)))))

(! (POSITION-UPDATE $H CL))

(! (PRINT-LIST (FORMAT NIL "~a secured" $SYS)))
```

rcs-leak-isol

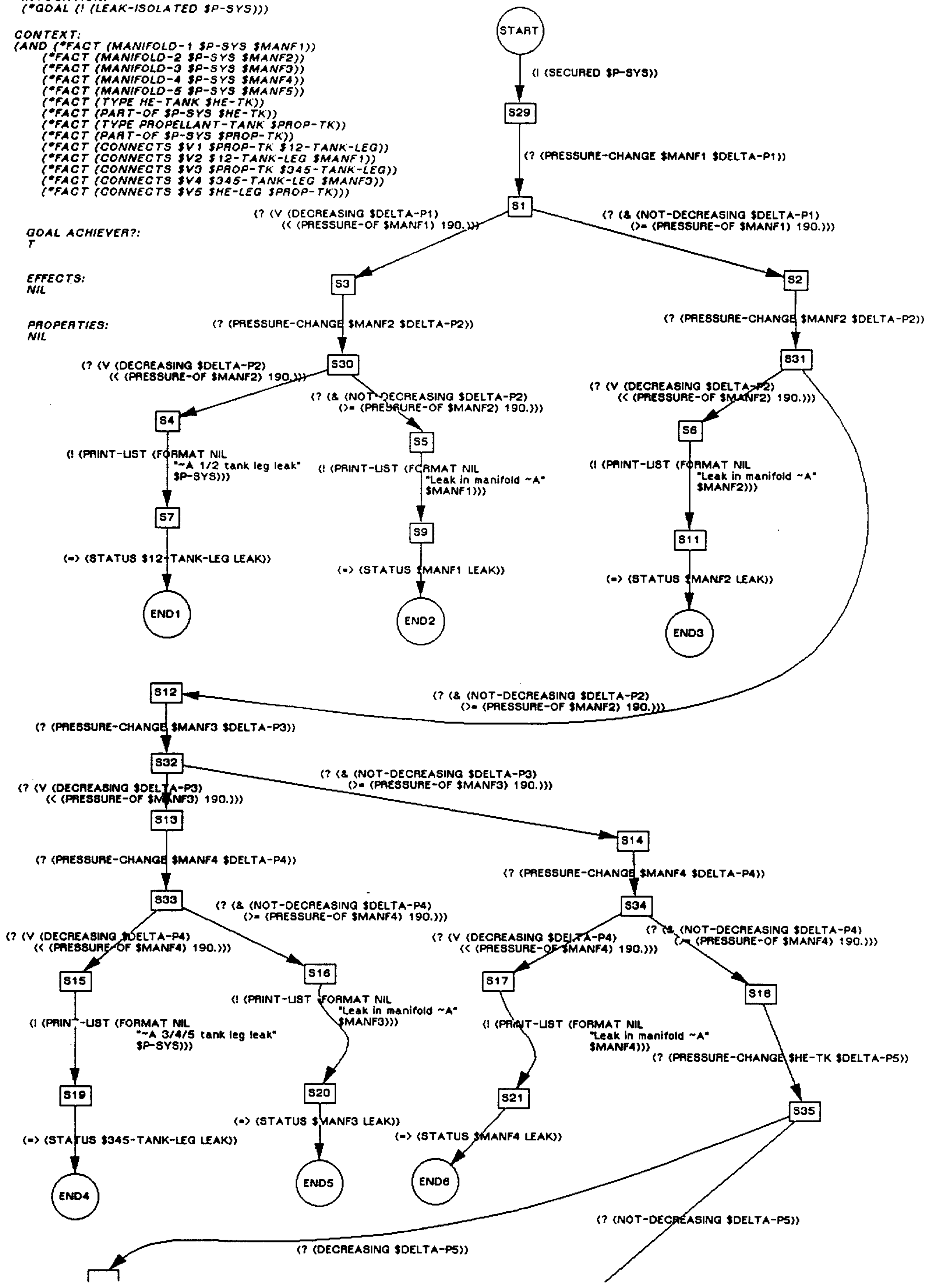
INVOCATION:
*(*GOAL (! (LEAK-ISOLATED $P-SYS)))*

CONTEXT:
```
(AND (*FACT (MANIFOLD-1 $P-SYS $MANF1))
     (*FACT (MANIFOLD-2 $P-SYS $MANF2))
     (*FACT (MANIFOLD-3 $P-SYS $MANF3))
     (*FACT (MANIFOLD-4 $P-SYS $MANF4))
     (*FACT (MANIFOLD-5 $P-SYS $MANF5))
     (*FACT (TYPE HE-TANK $HE-TK))
     (*FACT (PART-OF $P-SYS $HE-TK))
     (*FACT (TYPE PROPELLANT-TANK $PROP-TK))
     (*FACT (PART-OF $P-SYS $PROP-TK))
     (*FACT (CONNECTS $V1 $PROP-TK $12-TANK-LEG))
     (*FACT (CONNECTS $V2 $12-TANK-LEG $MANF1))
     (*FACT (CONNECTS $V3 $PROP-TK $345-TANK-LEG))
     (*FACT (CONNECTS $V4 $345-TANK-LEG $MANF3))
     (*FACT (CONNECTS $V5 $HE-LEG $PROP-TK)))
```

GOAL ACHIEVER?:
T

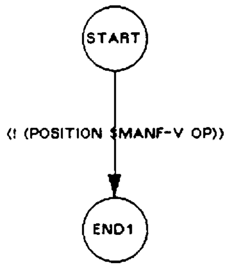
EFFECTS:
NIL

PROPERTIES:
NIL


START
(! (SECURED $P-SYS))
S29
(? (PRESSURE-CHANGE $MANF1 $DELTA-P1))
S1
(? (V (DECREASING $DELTA-P1) (< (PRESSURE-OF $MANF1) 190.)))
(? (& (NOT-DECREASING $DELTA-P1) (>= (PRESSURE-OF $MANF1) 190.)))
S3
S2
(? (PRESSURE-CHANGE $MANF2 $DELTA-P2))
(? (PRESSURE-CHANGE $MANF2 $DELTA-P2))
S30
S31
(? (V (DECREASING $DELTA-P2) (< (PRESSURE-OF $MANF2) 190.)))
(? (& (NOT-DECREASING $DELTA-P2) (>= (PRESSURE-OF $MANF2) 190.)))
(? (V (DECREASING $DELTA-P2) (< (PRESSURE-OF $MANF2) 190.)))
S4
S5
S6
(! (PRINT-LIST (FORMAT NIL "~A 1/2 tank leg leak" $P-SYS)))
(! (PRINT-LIST (FORMAT NIL "Leak in manifold ~A" $MANF1)))
(! (PRINT-LIST (FORMAT NIL "Leak in manifold ~A" $MANF2)))
S7
S9
S11
(=> (STATUS $12-TANK-LEG LEAK))
(=> (STATUS $MANF1 LEAK))
(=> (STATUS $MANF2 LEAK))
END1
END2
END3
S12
(? (& (NOT-DECREASING $DELTA-P2) (>= (PRESSURE-OF $MANF2) 190.)))
(? (PRESSURE-CHANGE $MANF3 $DELTA-P3))
S32
(? (& (NOT-DECREASING $DELTA-P3) (>= (PRESSURE-OF $MANF3) 190.)))
(? (V (DECREASING $DELTA-P3) (< (PRESSURE-OF $MANF3) 190.)))
S13
S14
(? (PRESSURE-CHANGE $MANF4 $DELTA-P4))
(? (PRESSURE-CHANGE $MANF4 $DELTA-P4))
S33
S34
(? (& (NOT-DECREASING $DELTA-P4) (>= (PRESSURE-OF $MANF4) 190.)))
(? (V (DECREASING $DELTA-P4) (< (PRESSURE-OF $MANF4) 190.)))
(? (V (DECREASING $DELTA-P4) (< (PRESSURE-OF $MANF4) 190.)))
(? (& (NOT-DECREASING $DELTA-P4) (>= (PRESSURE-OF $MANF4) 190.)))
S15
S16
S17
S18
(! (PRINT-LIST (FORMAT NIL "~A 3/4/5 tank leg leak" $P-SYS)))
(! (PRINT-LIST (FORMAT NIL "Leak in manifold ~A" $MANF3)))
(! (PRINT-LIST (FORMAT NIL "Leak in manifold ~A" $MANF4)))
(? (PRESSURE-CHANGE $HE-TK $DELTA-P5))
S19
S20
S21
S35
(=> (STATUS $345-TANK-LEG LEAK))
(=> (STATUS $MANF3 LEAK))
(=> (STATUS $MANF4 LEAK))
END4
END5
END6
(? (NOT-DECREASING $DELTA-P5))
(? (DECREASING $DELTA-P5))

override-manf-cl

```
INVOCATION:
(AND (*FACT (OVERRIDE $MANF CL-OVRD))
     (*FACT (STATUS $MANF GOOD))
     (*FACT (TYPE MANIFOLD $MANF))
     (*FACT (~ (UNDERPRESSURIZED $P-SYS)))
     (*FACT (PART-OF $P-SYS $MANF))
     (*FACT (CONNECTS $MANF-V $LEG $MANF)))
CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
(~ (OVERRIDE $MANF CL-OVRD))

PROPERTIES:
NIL
```



request-and-answer-pressure

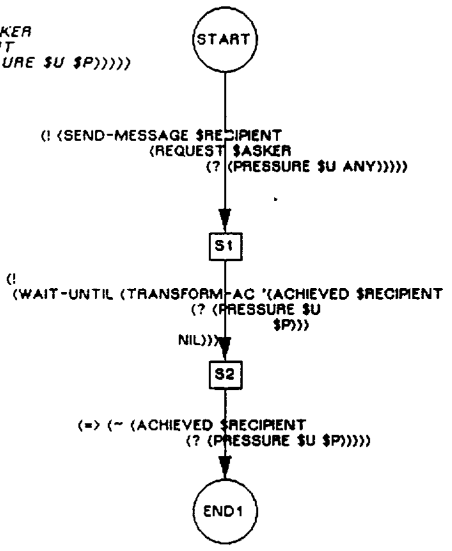
INVOCATION:
```
(*GOAL (! (REQUESTED $ASKER
                     $RECIPIENT
                     (? (PRESSURE $U $P)))))
```

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL



```
(! (SEND-MESSAGE $RECIPIENT
                 (REQUEST $ASKER
                          (? (PRESSURE $U ANY)))))

(!
 (WAIT-UNTIL (TRANSFORM-AC '(ACHIEVED $RECIPIENT
                                      (? (PRESSURE $U
                                                   $P)))
                           NIL)))

(=> (~ (ACHIEVED $RECIPIENT
                 (? (PRESSURE $U $P)))))
```

pressure-request

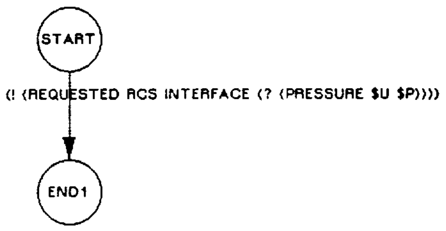
INVOCATION:
*(*GOAL (? (UPDATED-PRESSURE $U $P)))*

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
(PRESSURE $U $P)

PROPERTIES:
NIL

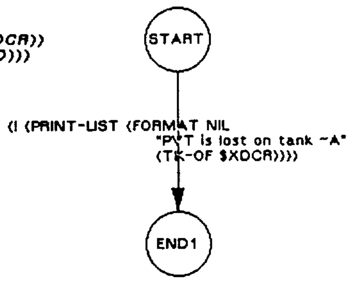


pvt-lost

INVOCATION:
```
(AND (*FACT (TYPE P-XDCR $XDCR))
     (*FACT (STATUS $XDCR BAD)))
```

CONTEXT:
NIL

GOAL ACHIEVER?:
T



```
(! (PRINT-LIST (FORMAT NIL
                 "PVT is lost on tank ~A"
                 (TK-OF $XDCR))))
```

EFFECTS:
```
(STATUS (PVT (TK-OF $XDCR)) BAD)
```

PROPERTIES:
NIL

switch-update

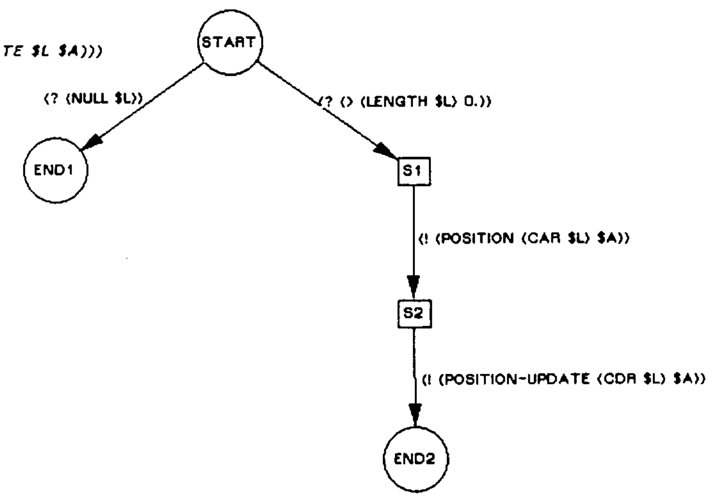
INVOCATION:
*(*GOAL (! (POSITION-UPDATE $L $A)))*

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL



quantity1

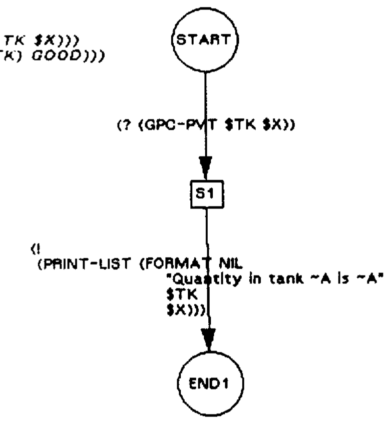
INVOCATION:
```
(AND (*GOAL (? (QUANTITY $TK $X)))
     (*FACT (STATUS (PVT $TK) GOOD)))
```

CONTEXT:
NIL

GOAL ACHIEVER?:
T

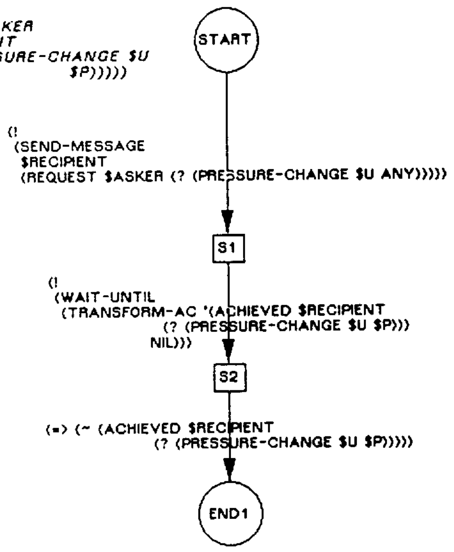
EFFECTS:
(QUANTITY $TK $X)

PROPERTIES:
NIL



request-and-answer-pressure-change

```
INVOCATION:
(*GOAL (! (REQUESTED $ASKER
                     $RECIPIENT
                     (? (PRESSURE-CHANGE $U
                                         $P)))))

CONTEXT:
NIL

 GOAL ACHIEVER?:
 T

EFFECTS:
NIL

PROPERTIES:
NIL
```



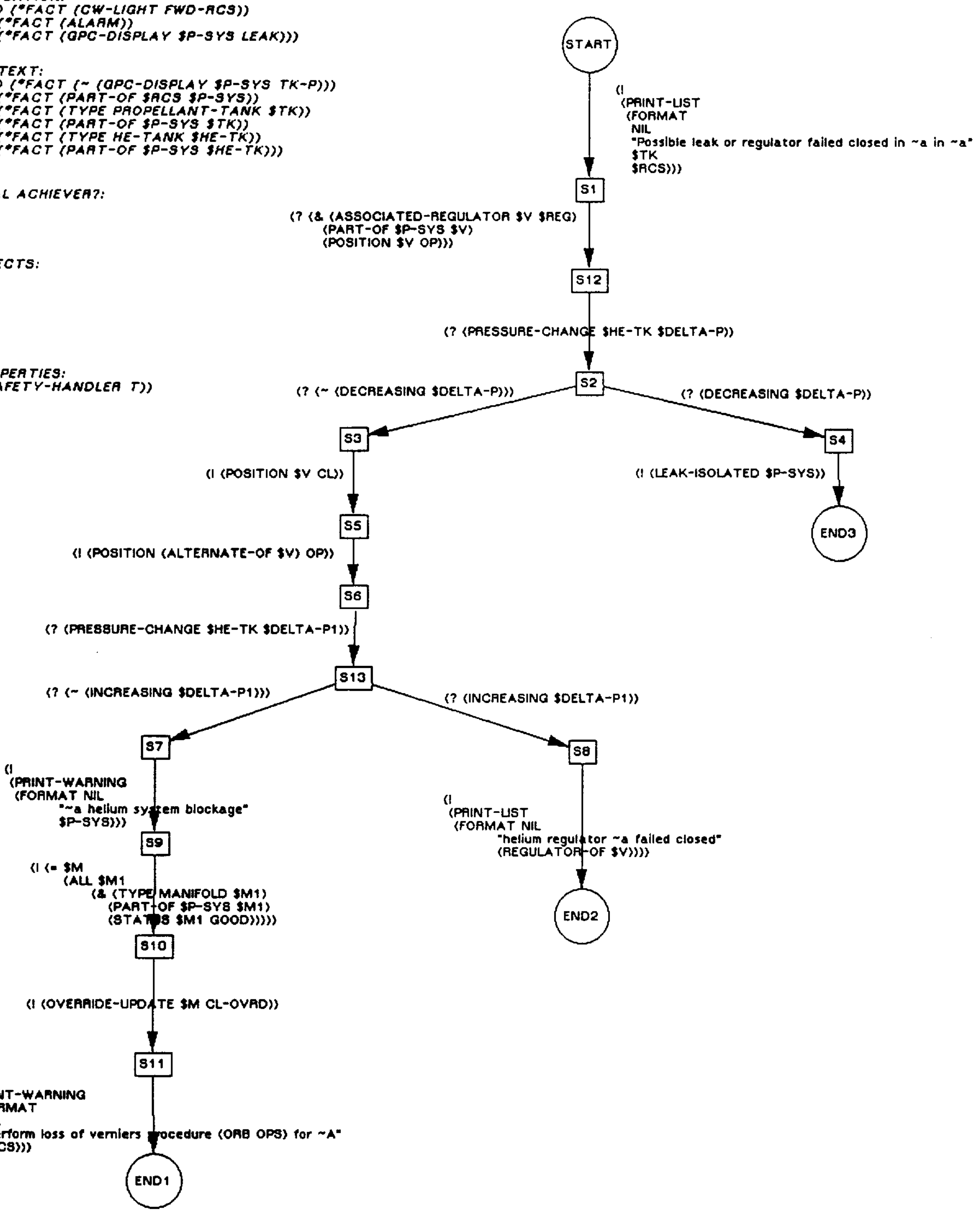
```
(!
 (SEND-MESSAGE
  $RECIPIENT
  (REQUEST $ASKER (? (PRESSURE-CHANGE $U ANY)))))
```

```
(!
 (WAIT-UNTIL
  (TRANSFORM-AC '(ACHIEVED $RECIPIENT
                  (? (PRESSURE-CHANGE $U $P)))
                NIL)))
```

```
(=) (~ (ACHIEVED $RECIPIENT
              (? (PRESSURE-CHANGE $U $P)))))
```

cw-rcs-prop-low

```
INVOCATION:
(AND (*FACT (CW-LIGHT FWD-RCS))
     (*FACT (ALARM))
     (*FACT (GPC-DISPLAY $P-SYS LEAK)))

CONTEXT:
(AND (*FACT (~ (GPC-DISPLAY $P-SYS TK-P)))
     (*FACT (PART-OF $RCS $P-SYS))
     (*FACT (TYPE PROPELLANT-TANK $TK))
     (*FACT (PART-OF $P-SYS $TK))
     (*FACT (TYPE HE-TANK $HE-TK))
     (*FACT (PART-OF $P-SYS $HE-TK)))

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
((SAFETY-HANDLER T))
```



START
(! (PRINT-LIST (FORMAT NIL "Possible leak or regulator failed closed in ~a in ~a" $TK $RCS)))
S1
(? (& (ASSOCIATED-REGULATOR $V $REG) (PART-OF $P-SYS $V) (POSITION $V OP)))
S12
(? (PRESSURE-CHANGE $HE-TK $DELTA-P))
S2
(? (~ (DECREASING $DELTA-P)))
(? (DECREASING $DELTA-P))
S3
S4
(! (POSITION $V CL))
(! (LEAK-ISOLATED $P-SYS))
END3
S5
(! (POSITION (ALTERNATE-OF $V) OP))
S6
(? (PRESSURE-CHANGE $HE-TK $DELTA-P1))
S13
(? (~ (INCREASING $DELTA-P1)))
(? (INCREASING $DELTA-P1))
S7
S8
(! (PRINT-WARNING (FORMAT NIL "~a helium system blockage" $P-SYS)))
(! (PRINT-LIST (FORMAT NIL "helium regulator ~a failed closed" (REGULATOR-OF $V))))
S9
END2
(! (= $M (ALL $M1 (& (TYPE MANIFOLD $M1) (PART-OF $P-SYS $M1) (STATUS $M1 GOOD)))))
S10
(! (OVERRIDE-UPDATE $M CL-OVRD))
S11
(! (PRINT-WARNING (FORMAT NIL "Perform loss of verniers procedure (ORB OPS) for ~A" $RCS)))
END1

gpc-pvt

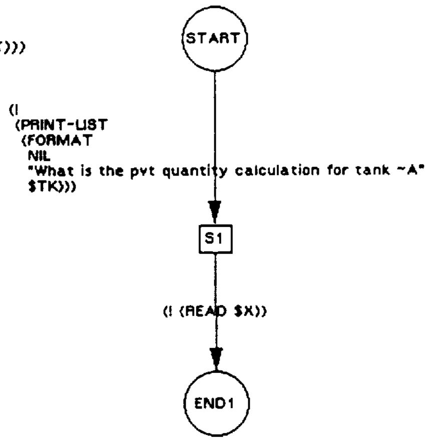
INVOCATION:
*(*GOAL (? (GPC-PVT $TK $X)))*

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL



dap-as-required

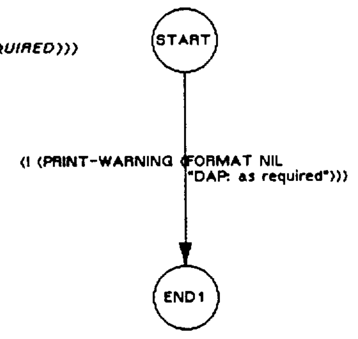
INVOCATION:
*(*GOAL (! (STATUS DAP AS-REQUIRED)))*

CONTEXT:
NIL

GOAL ACHIEVER?:
T

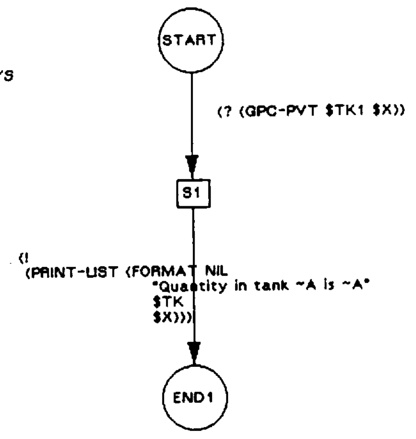
EFFECTS:
(STATUS DAP AS-REQUIRED)

PROPERTIES:
NIL



quantity2

```
INVOCATION:
(AND (*GOAL (? (QUANTITY $TK $X)))
     (*FACT (TYPE PROPELLANT-TANK $TK))
     (*FACT (STATUS (PVT $TK) BAD))
     (*FACT (PART-OF $P-SYS $TK))
     (*FACT (OTHER-PROPELLANT-SYSTEM $P-SYS
                                     $P-SYS1))
     (*FACT (TYPE PROPELLANT-TANK $TK1))
     (*FACT (PART-OF $P-SYS1 $TK1))
CONTEXT:
NIL  (*FACT (STATUS (PVT $TK1) GOOD)))


GOAL ACHIEVER?:
T


EFFECTS:
(QUANTITY $TK $X)


PROPERTIES:
NIL
```



dap-free-drift

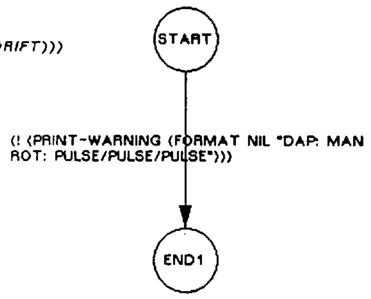
INVOCATION:
*(*GOAL (! (STATUS DAP FREE-DRIFT)))*

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
(STATUS DAP FREE-DRIFT)

PROPERTIES:
NIL


START
(! (PRINT-WARNING (FORMAT NIL "DAP: MAN ROT: PULSE/PULSE/PULSE")))
END1

alarm-off-underp

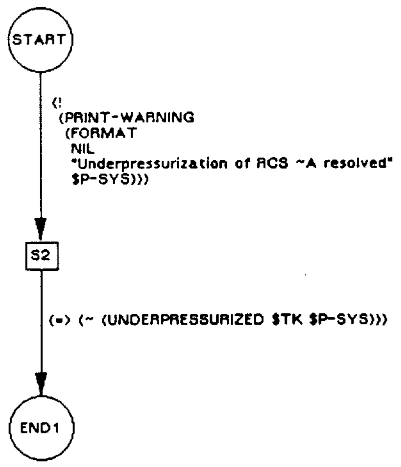
INVOCATION:
*(AND (*FACT (~ (ALARM)))*
*(*FACT (UNDERPRESSURIZED $TK $P-SYS)))*

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL


START
(!
(PRINT-WARNING
(FORMAT
NIL
"Underpressurization of RCS ~A resolved"
$P-SYS)))
S2
(=) (~ (UNDERPRESSURIZED $TK $P-SYS)))
END1

open-or-close

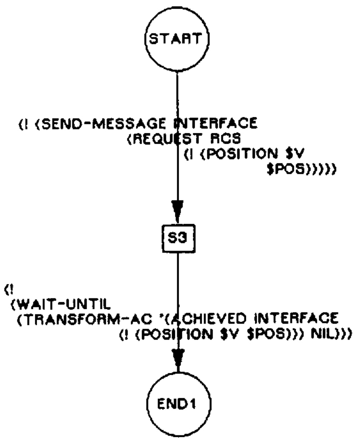
INVOCATION:
*(*GOAL (! (POSITION $V $POS)))*

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
(POSITION $V $POS)

PROPERTIES:
NIL


START
(! (SEND-MESSAGE INTERFACE (REQUEST RCS (! (POSITION $V $POS)))))
S3
(! (WAIT-UNTIL (TRANSFORM-AC '(ACHIEVED INTERFACE (! (POSITION $V $POS))) NIL)))
END1

override-update

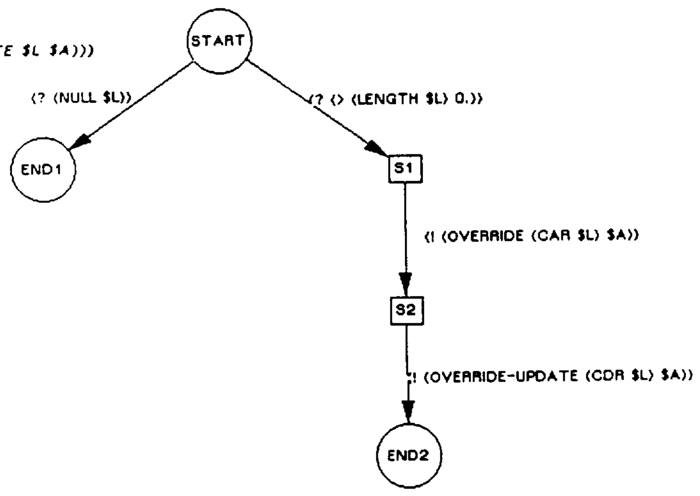
INVOCATION:
*(*GOAL (! (OVERRIDE-UPDATE $L $A)))*

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL



override

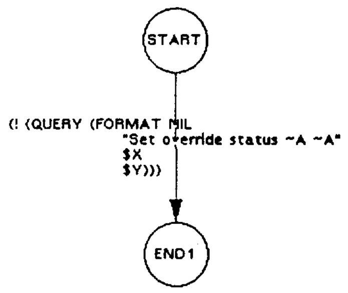
INVOCATION:
*(*GOAL (! (OVERRIDE $X $Y)))*

CONTEXT:
NIL

GOAL ACHIEVER?:
T

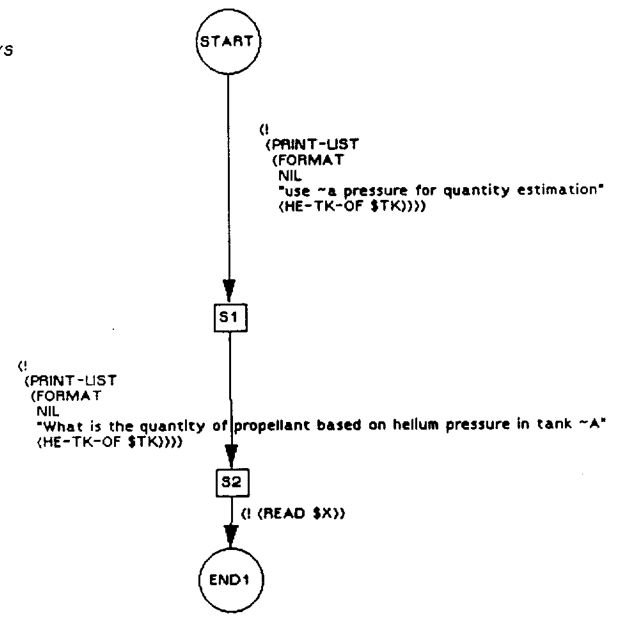
EFFECTS:
(OVERRIDE $X $Y)

PROPERTIES:
NIL



quantity3

```
INVOCATION:
(AND (*GOAL (? (QUANTITY $TK $X)))
     (*FACT (TYPE PROPELLANT-TANK $TK))
     (*FACT (STATUS (PVT $TK) BAD))
     (*FACT (PART-OF $P-SYS $TK))
     (*FACT (OTHER-PROPELLANT-SYSTEM $P-SYS
                                     $P-SYS1))
     (*FACT (TYPE PROPELLANT-TANK $TK1))
CONTEXT:(*FACT (PART-OF $P-SYS1 $TK1))
NIL  (*FACT (STATUS (PVT $TK1) BAD)))


GOAL ACHIEVER?:
T


EFFECTS:
NIL


PROPERTIES:
NIL
```



```
(!
 (PRINT-LIST
  (FORMAT
   NIL
   "use ~a pressure for quantity estimation"
   (HE-TK-OF $TK))))
```

```
(!
 (PRINT-LIST
  (FORMAT
   NIL
   "What is the quantity of propellant based on helium pressure in tank ~A"
   (HE-TK-OF $TK))))
```

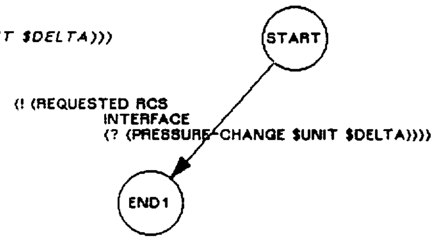
```
(! (READ $X))
```

pressure-change

INVOCATION:
*(*GOAL (? (PRESSURE-CHANGE $UNIT $DELTA)))*

CONTEXT:
NIL

GOAL ACHIEVER?:
T

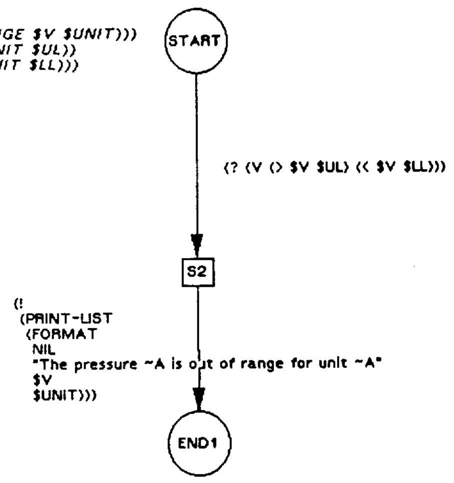


EFFECTS:
NIL

PROPERTIES:
NIL

out-of-range

```
INVOCATION:
(AND (*GOAL (? (OUT-OF-RANGE $V $UNIT)))
     (*FACT (PRESSURE-UL $UNIT $UL))
     (*FACT (PRESSURE-LL $UNIT $LL)))

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL
```



```
(? (V (> $V $UL) (< $V $LL)))
```

```
(!
 (PRINT-LIST
  (FORMAT
   NIL
   "The pressure ~A is out of range for unit ~A"
   $V
   $UNIT)))
```

alarm-off-overp

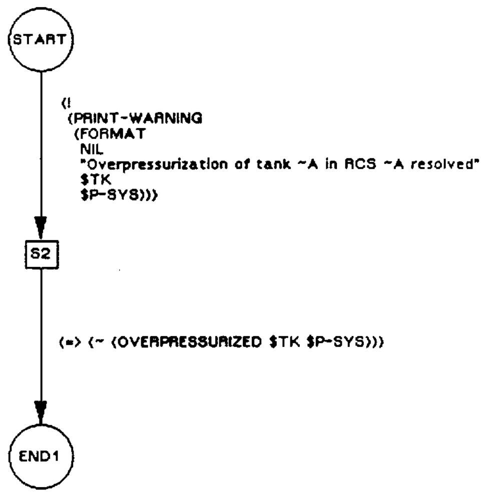
INVOCATION:
```
(AND (*FACT (~ (ALARM)))
     (*FACT (OVERPRESSURIZED $TK $P-SYS)))
```

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL

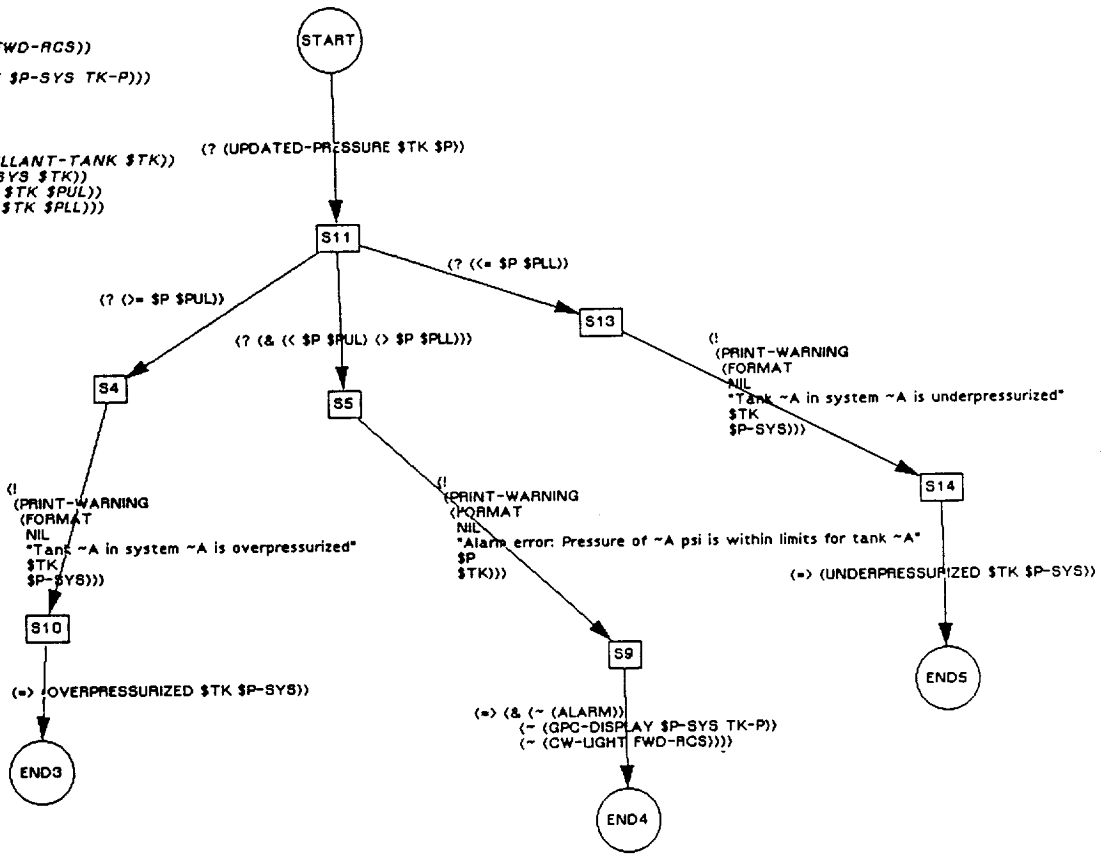


```
(!
 (PRINT-WARNING
  (FORMAT
   NIL
   "Overpressurization of tank ~A in RCS ~A resolved"
   $TK
   $P-SYS)))
```

```
(=> (~ (OVERPRESSURIZED $TK $P-SYS)))
```

cw-rcs-alarm

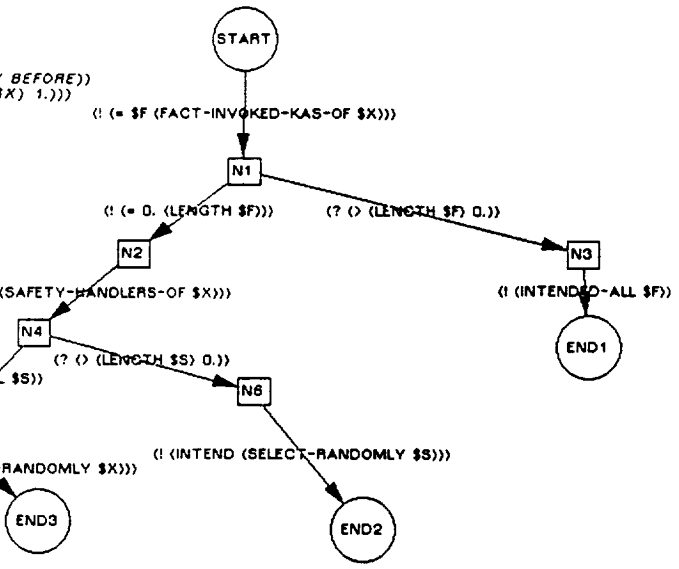
```
INVOCATION:
(AND (*FACT (CW-LIGHT FWD-RCS))
     (*FACT (ALARM))
     (*FACT (GPC-DISPLAY $P-SYS TK-P)))


CONTEXT:
(AND (*FACT (TYPE PROPELLANT-TANK $TK))
     (*FACT (PART-OF $P-SYS $TK))
     (*FACT (PRESSURE-UL $TK $PUL))
     (*FACT (PRESSURE-LL $TK $PLL)))

    GOAL ACHIEVER?:
    T

    EFFECTS:
    NIL


    PROPERTIES:
    ((SAFETY-HANDLER T))
```



```
(? (UPDATED-PRESSURE $TK $P))

(? (>= $P $PUL))

(? (& (< $P $PUL) (> $P $PLL)))

(? (<= $P $PLL))

(!
 (PRINT-WARNING
  (FORMAT
   NIL
   "Tank ~A in system ~A is overpressurized"
   $TK
   $P-SYS)))

(=> (OVERPRESSURIZED $TK $P-SYS))

(!
 (PRINT-WARNING
  (FORMAT
   NIL
   "Alarm error: Pressure of ~A psi is within limits for tank ~A"
   $P
   $TK)))

(=> (& (~ (ALARM))
       (~ (GPC-DISPLAY $P-SYS TK-P))
       (~ (CW-LIGHT FWD-RCS))))

(!
 (PRINT-WARNING
  (FORMAT
   NIL
   "Tank ~A in system ~A is underpressurized"
   $TK
   $P-SYS)))

(=> (UNDERPRESSURIZED $TK $P-SYS))
```

selector

```
INVOCATION:
(*FACT (SOAK $X))

CONTEXT:
(AND (*FACT (NOSAFETY BEFORE))
     (*FACT (> (LENGTH $X) 1.)))

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL
```


START
(! (= $F (FACT-INVOKED-KAS-OF $X)))
N1
(! (= 0. (LENGTH $F)))
(? (> (LENGTH $F) 0.))
N2
N3
(! (= $S (SAFETY-HANDLERS-OF $X)))
(! (INTENDED-ALL $F))
N4
END1
(? (> (LENGTH $S) 0.))
(? (NULL $S))
N6
N5
(! (INTEND (SELECT-RANDOMLY $S)))
(! (INTEND (SELECT-RANDOMLY $X)))
END3
END2

selector2

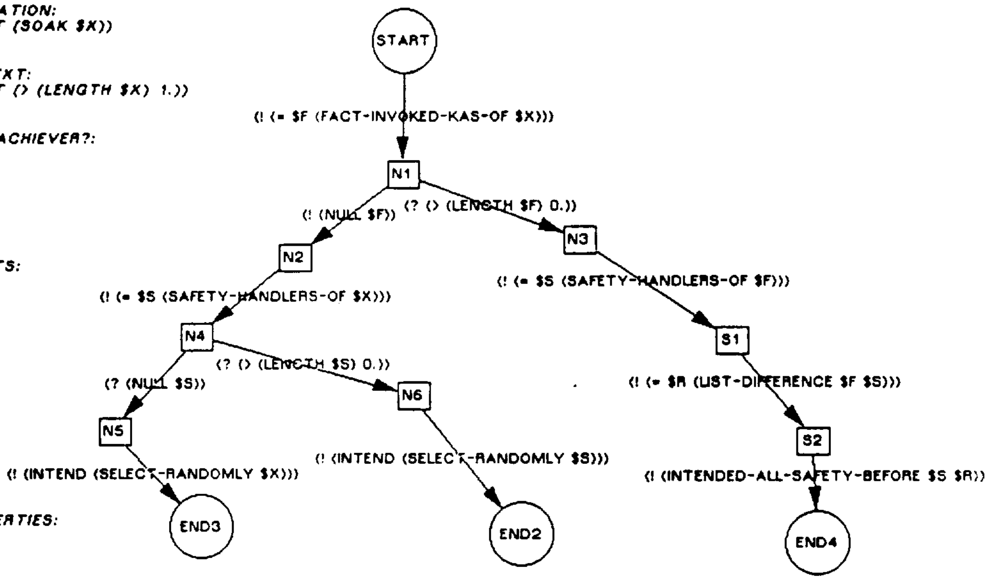
INVOCATION:
*(*FACT (SOAK $X))*

CONTEXT:
*(*FACT (> (LENGTH $X) 1.))*

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL


START
(! (= $F (FACT-INVOKED-KAS-OF $X)))
N1
(! (NULL $F))
(? (> (LENGTH $F) 0.))
N2
N3
(! (= $S (SAFETY-HANDLERS-OF $X)))
(! (= $S (SAFETY-HANDLERS-OF $F)))
N4
S1
(? (> (LENGTH $S) 0.))
(? (NULL $S))
N6
(! (= $R (LIST-DIFFERENCE $F $S)))
N5
S2
(! (INTEND (SELECT-RANDOMLY $S)))
(! (INTEND (SELECT-RANDOMLY $X)))
(! (INTENDED-ALL-SAFETY-BEFORE $S $R))
END3
END2
END4

Appendix C

The INTERFACE Database

The Interface database is much the same as the RCS database, except that it has knowledge of transducer readings and switch and talkback positions. The interface has no knowledge of pressures – it has to deduce these afresh from the latest transducer readings. The major differences are given below:

following values are op, cl, gpc

```
(position frcs-prop-tk-isol-12-switch op)
(position frcs-prop-tk-isol-345-switch op)
(position frcs-he-tk-isol-A-switch gpc)
(position frcs-he-tk-isol-B-switch gpc)
(position frcs-manf-1-switch op)
(position frcs-manf-2-switch op)
(position frcs-manf-3-switch op)
(position frcs-manf-4-switch op)
(position frcs-manf-5-switch op)
```

following values are op, cl, bp

```
(position frcs-prop-tk-isol-12-talkback op)
(position frcs-prop-tk-isol-345-talkback op)
(position frcs-he-tk-isol-A-talkback op)
(position frcs-he-tk-isol-B-talkback cl)
(position frcs-manf-1-talkback op)
(position frcs-manf-2-talkback op)
(position frcs-manf-3-talkback op)
(position frcs-manf-4-talkback op)
(position frcs-manf-5-talkback op)
```

```
(mode rcs (status \$unit bad) always)

(~ (alarm))
```

Appendix D

The INTERFACE Procedures

The various KAs used in the `INTERFACE` KA library are given on the following pages.

talkback-bp

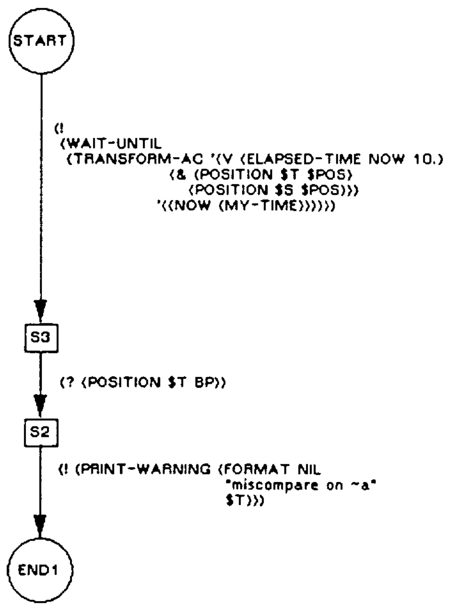
INVOCATION:
*(*FACT (POSITION $T BP))*

CONTEXT:
*(*FACT (ASSOCIATED-TALKBACK $S $T))*

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL



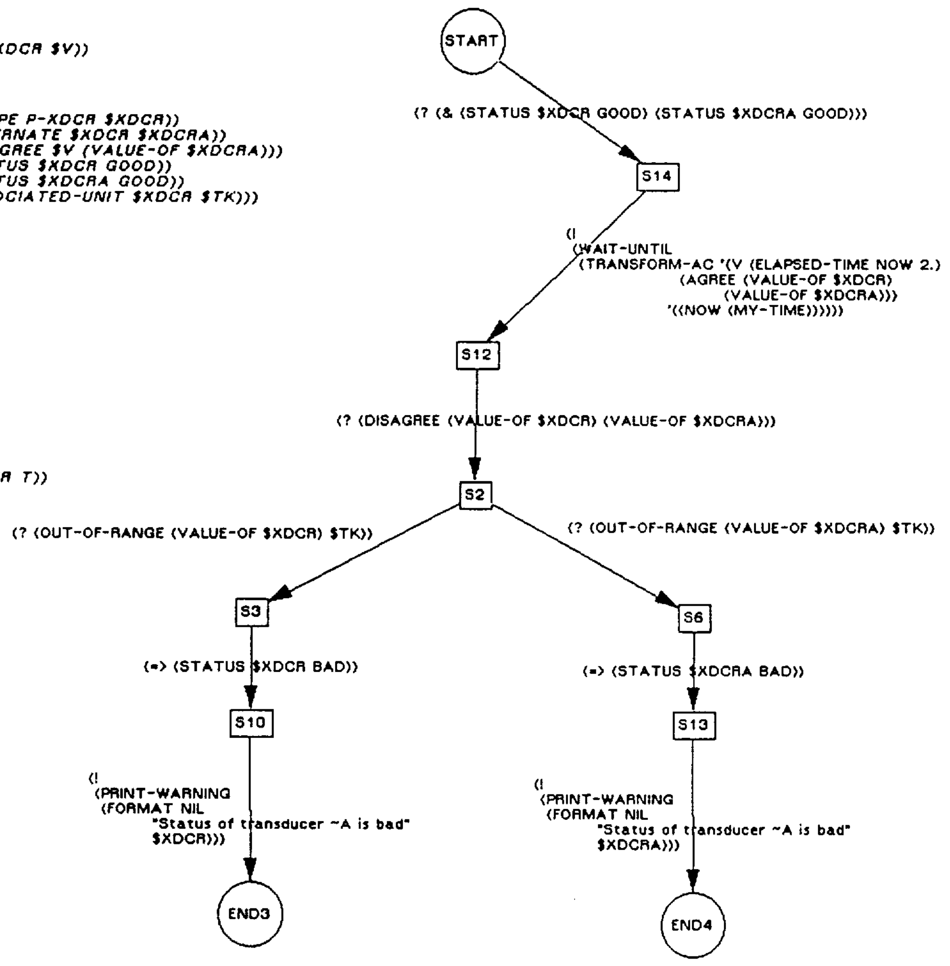
```
(!
 (WAIT-UNTIL
  (TRANSFORM-AC '(V (ELAPSED-TIME NOW 10.)
                 (& (POSITION $T $POS)
                    (POSITION $S $POS)))
                '((NOW (MY-TIME))))))
```

```
(? (POSITION $T BP))
```

```
(! (PRINT-WARNING (FORMAT NIL
                          "miscompare on ~a"
                          $T)))
```

xdcr-bad

```
INVOCATION:
(*FACT (VALUE $XDCR $V))

CONTEXT:
(AND (*FACT (TYPE P-XDCR $XDCR))
     (*FACT (ALTERNATE $XDCR $XDCRA))
     (*FACT (DISAGREE $V (VALUE-OF $XDCRA)))
     (*FACT (STATUS $XDCR GOOD))
     (*FACT (STATUS $XDCRA GOOD))
     (*FACT (ASSOCIATED-UNIT $XDCR $TK)))

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
((SAFETY-HANDLER T))
```



```
(? (& (STATUS $XDCR GOOD) (STATUS $XDCRA GOOD)))

(!
 (WAIT-UNTIL
  (TRANSFORM-AC '(V (ELAPSED-TIME NOW 2.)
                    (AGREE (VALUE-OF $XDCR)
                           (VALUE-OF $XDCRA)))
                '((NOW (MY-TIME))))))

(? (DISAGREE (VALUE-OF $XDCR) (VALUE-OF $XDCRA)))

(? (OUT-OF-RANGE (VALUE-OF $XDCR) $TK))

(? (OUT-OF-RANGE (VALUE-OF $XDCRA) $TK))

(=> (STATUS $XDCR BAD))

(=> (STATUS $XDCRA BAD))

(!
 (PRINT-WARNING
  (FORMAT NIL
          "Status of transducer ~A is bad"
          $XDCR)))

(!
 (PRINT-WARNING
  (FORMAT NIL
          "Status of transducer ~A is bad"
          $XDCRA)))
```

switch-dilemma-1

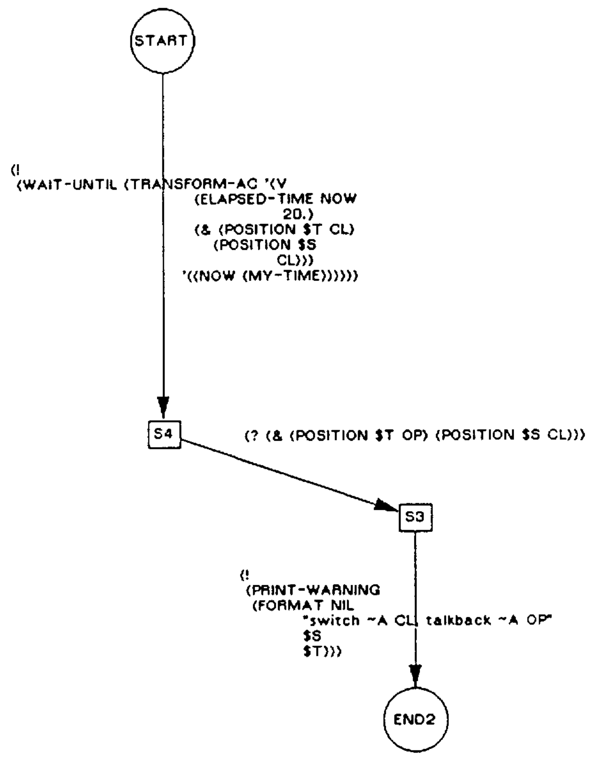
INVOCATION:
```
(AND (*FACT (POSITION $T OP))
     (*FACT (POSITION $S CL)))
```

CONTEXT:
```
(*FACT (ASSOCIATED-TALKBACK $S $T))
```

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL


START
(! (WAIT-UNTIL (TRANSFORM-AC '(V (ELAPSED-TIME NOW 20.) (& (POSITION $T CL) (POSITION $S CL))) '((NOW (MY-TIME))))))
S4
(? (& (POSITION $T OP) (POSITION $S CL)))
S3
(! (PRINT-WARNING (FORMAT NIL "switch ~A CL talkback ~A OP" $S $T)))
END2

open-or-close-valve

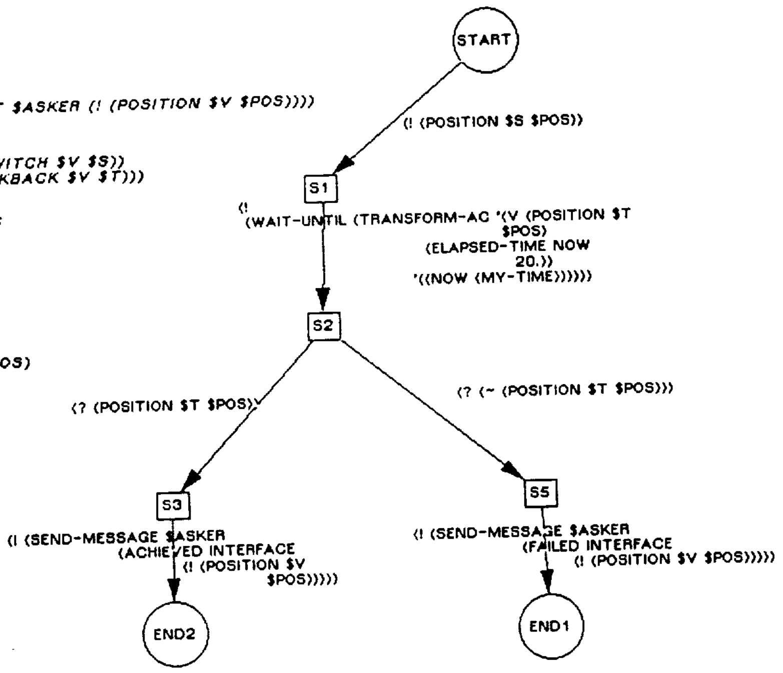
INVOCATION:
*(*FACT (REQUEST $ASKER (! (POSITION $V $POS))))*

CONTEXT:
*(AND (*FACT (SWITCH $V $S))*
*(*FACT (TALKBACK $V $T)))*

GOAL ACHIEVER?:
T

EFFECTS:
(POSITION $V $POS)

PROPERTIES:
NIL


START
(! (POSITION $S $POS))
S1
(! (WAIT-UNTIL (TRANSFORM-AC '(V (POSITION $T $POS) (ELAPSED-TIME NOW 20.)) '((NOW (MY-TIME))))))
S2
(? (POSITION $T $POS))
(? (~ (POSITION $T $POS)))
S3
S5
(! (SEND-MESSAGE $ASKER (ACHIEVED INTERFACE (! (POSITION $V $POS)))))
(! (SEND-MESSAGE $ASKER (FAILED INTERFACE (! (POSITION $V $POS)))))
END2
END1

open-or-close-switch

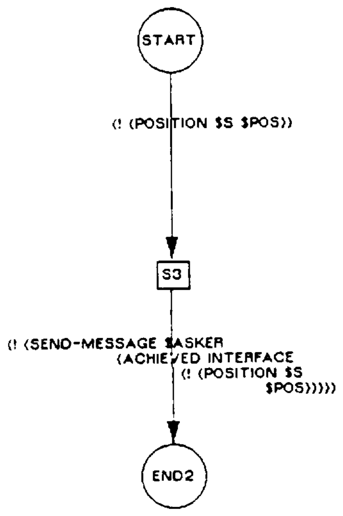
INVOCATION:
*(*FACT (REQUEST $ASKER (! (POSITION $S $POS))))*

CONTEXT:
*(*FACT (TYPE SWITCH $S))*

GOAL ACHIEVER?:
T

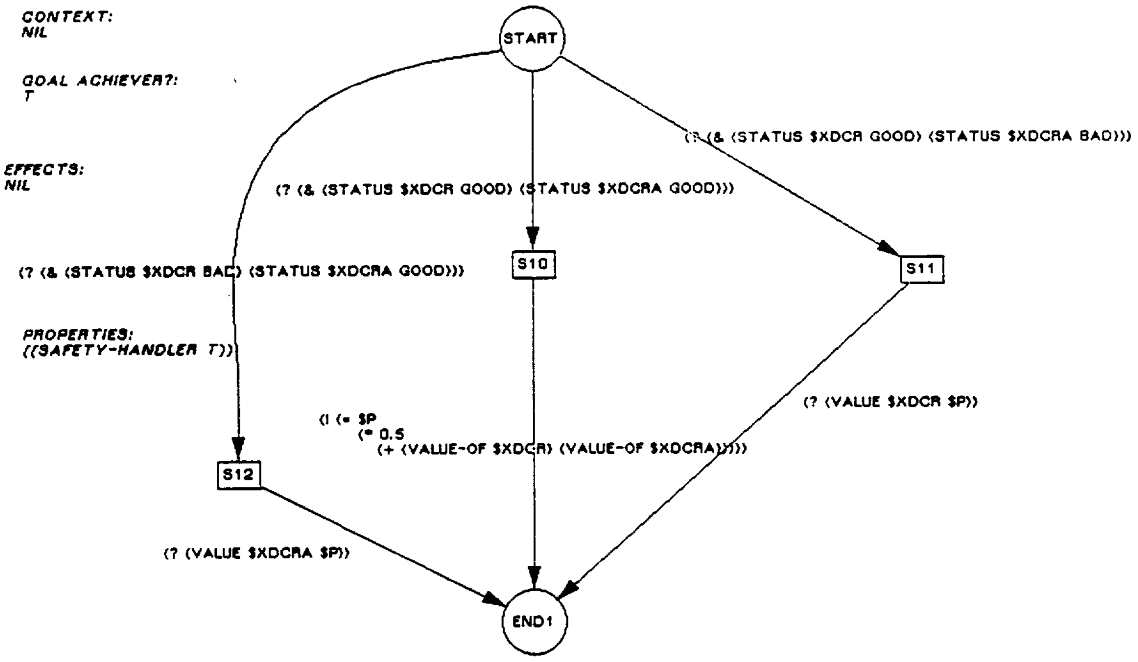
EFFECTS:
(POSITION $S $POS)

PROPERTIES:
NIL


START
(! (POSITION $S $POS))
S3
(! (SEND-MESSAGE $ASKER (ACHIEVED INTERFACE (! (POSITION $S $POS)))))
END2

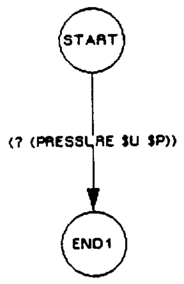
pressure

```
INVOCATION:
(AND (*GOAL (? (PRESSURE $TK $P)))
     (*FACT (ASSOCIATED-UNIT $XDCR $TK))
     (*FACT (TYPE P-XDCR $XDCR))
     (*FACT (ALTERNATE $XDCR $XDCRA)))

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
((SAFETY-HANDLER T))
```


START
(? (& (STATUS $XDCR GOOD) (STATUS $XDCRA BAD)))
(? (& (STATUS $XDCR GOOD) (STATUS $XDCRA GOOD)))
(? (& (STATUS $XDCR BAD) (STATUS $XDCRA GOOD)))
S10
S11
S12
(? (VALUE $XDCR $P))
(! (= $P
(* 0.5
(+ (VALUE-OF $XDCR) (VALUE-OF $XDCRA)))))
(? (VALUE $XDCRA $P))
END1

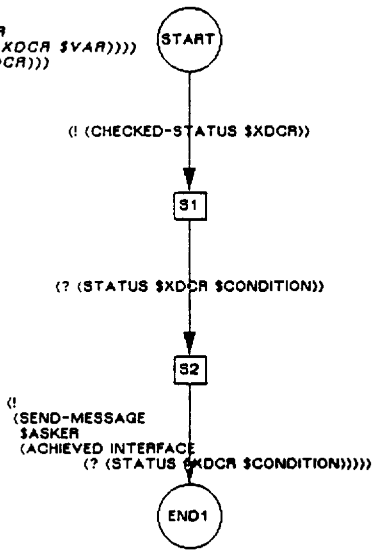
pressure-upstream

```
INVOCATION:
(AND (*GOAL (? (PRESSURE $TK $P)))
     (*FACT (CONNECTS $V $U $TK))
     (*FACT (POSITION $V OP)))

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL
```



xdcr-status-request

```
INVOCATION:
(AND (*FACT (REQUEST $ASKER
                      (? (STATUS $XDCR $VAR))))
     (*FACT (TYPE P-XDCR $XDCR)))

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL
```


START
(! (CHECKED-STATUS $XDCR))
S1
(? (STATUS $XDCR $CONDITION))
S2
(! (SEND-MESSAGE $ASKER (ACHIEVED INTERFACE (? (STATUS $XDCR $CONDITION)))))
END1

check-xdcr

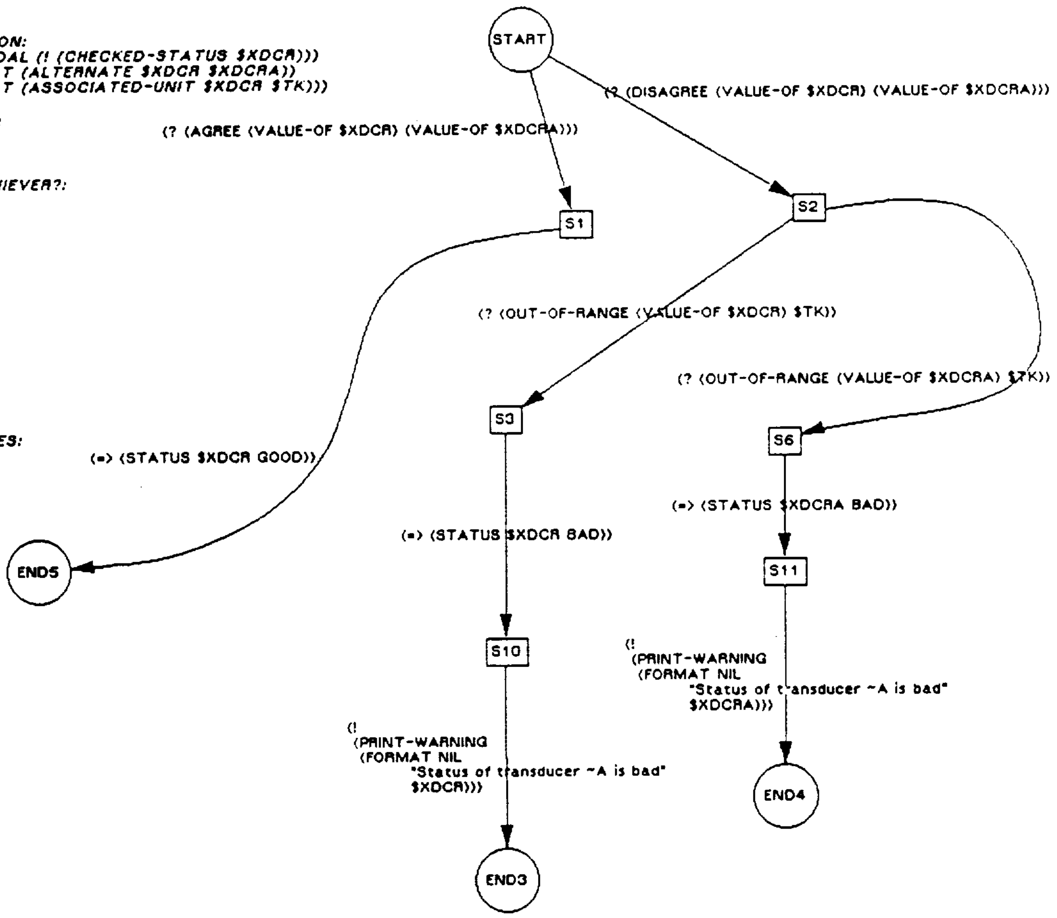
INVOCATION:
*(AND (*GOAL (! (CHECKED-STATUS $XDCR)))*
*(*FACT (ALTERNATE $XDCR $XDCRA))*
*(*FACT (ASSOCIATED-UNIT $XDCR $TK)))*

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL



START
(? (DISAGREE (VALUE-OF $XDCR) (VALUE-OF $XDCRA)))
(? (AGREE (VALUE-OF $XDCR) (VALUE-OF $XDCRA)))
S1
S2
(? (OUT-OF-RANGE (VALUE-OF $XDCR) $TK))
(? (OUT-OF-RANGE (VALUE-OF $XDCRA) $TK))
S3
S6
(=) (STATUS $XDCR GOOD))
(=) (STATUS $XDCRA BAD))
(=) (STATUS $XDCR BAD))
END5
S11
S10
(! (PRINT-WARNING (FORMAT NIL "Status of transducer ~A is bad" $XDCRA)))
(! (PRINT-WARNING (FORMAT NIL "Status of transducer ~A is bad" $XDCR)))
END4
END3

request-monitor

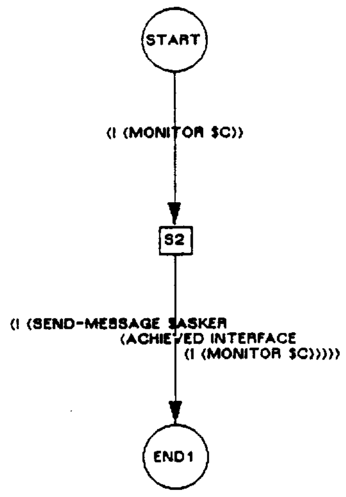
INVOCATION:
*(*FACT (REQUEST $ASKER (! (MONITOR $C))))*

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL



advise

```
INVOCATION:
(*FACT (REQUEST $ASKER
                (! (ADVISE $X $PERIOD))))

CONTEXT:
NIL

GOAL ACHIEVER?:
T


EFFECTS:
(MODE $ASKER $X $PERIOD)


PROPERTIES:
NIL
```

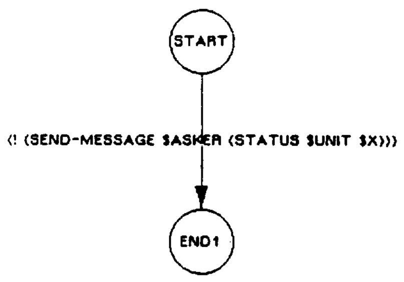
status-always

INVOCATION:
```
(AND (*FACT (MODE $ASKER
                  (STATUS $UNIT $X)
                  ALWAYS))
     (*FACT (STATUS $UNIT $X)))
```
CONTEXT:
NIL

GOAL ACHIEVER?:
T

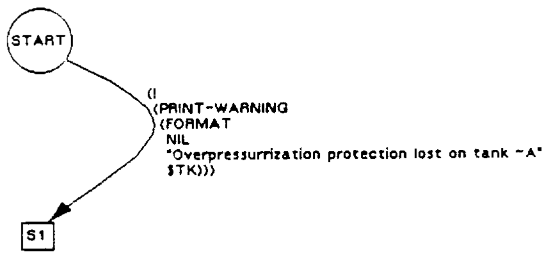
EFFECTS:
NIL

PROPERTIES:
NIL



lost-auto-close

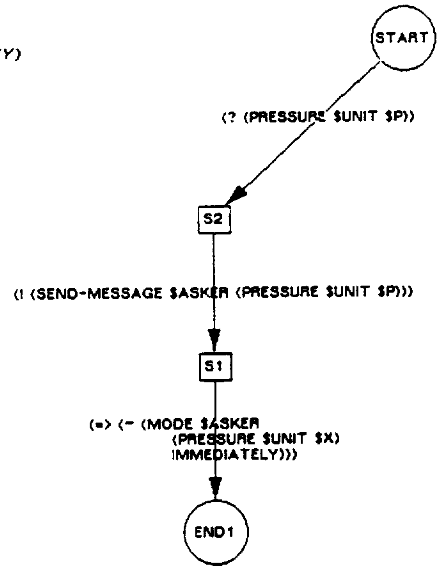
```
INVOCATION:
(AND (*FACT (STATUS $XDCR BAD))
     (*FACT (TYPE P-XDCR $XDCR))
     (*FACT (ALARM-INITIATOR $XDCR))
     (*FACT (ASSOCIATED-UNIT $XDCR $TK)))

CONTEXT:
NIL

GOAL ACHIEVER?:
T
```



```
EFFECTS:
(STATUS (OVERPRESSURIZATION-PROTECTION-OF $TK)
        LOST)

PROPERTIES:
NIL
```

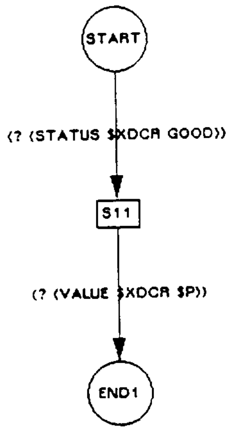
pressure-immediately

```
INVOCATION:
(*FACT (MODE $ASKER
            (PRESSURE $UNIT ANY)
            IMMEDIATELY))

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL
```



pressure-no-alt

```
INVOCATION:
(AND (*GOAL (? (PRESSURE $TK $P)))
     (*FACT (ASSOCIATED-UNIT $XDCR $TK))
     (*FACT (TYPE P-XDCR $XDCR))
     (*FACT (STATUS $XDCR GOOD))
     (*FACT (~ (ALTERNATE $XDCR $XDCRA))))
CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
((SAFETY-HANDLER T))
```



switch-valve

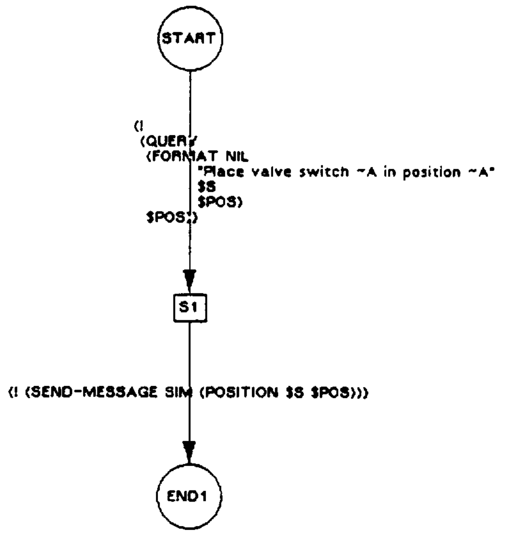
INVOCATION:
```
(AND (*GOAL (! (POSITION $S $POS)))
     (*FACT (TYPE SWITCH $S)))
```

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
```
(POSITION $S $POS)
```

PROPERTIES:
NIL

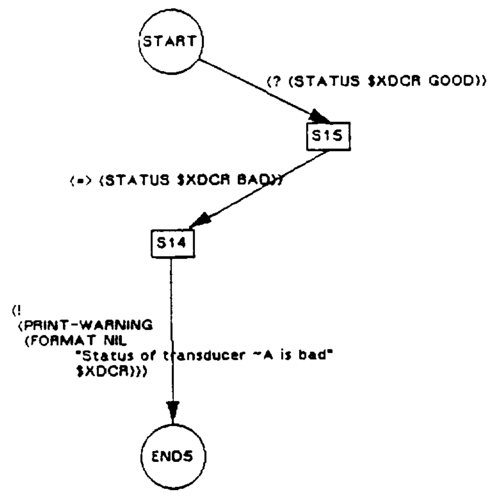


```
(!
 (QUERY
  (FORMAT NIL
          "Place valve switch ~A in position ~A"
          $S
          $POS)
  $POS))
```

```
(! (SEND-MESSAGE SIM (POSITION $S $POS)))
```

xdcr-bad-0-reading

```
INVOCATION:
(AND (*FACT (VALUE $XDCR $V))
     (*FACT (TYPE P-XDCR $XDCR))
     (*FACT ((<= $V 0.))
     (*FACT (STATUS $XDCR GOOD)))

CONTEXT:
NIL


GOAL ACHIEVER?:
T




EFFECTS:
NIL




PROPERTIES:
((SAFETY-HANDLER T))
```



monitor

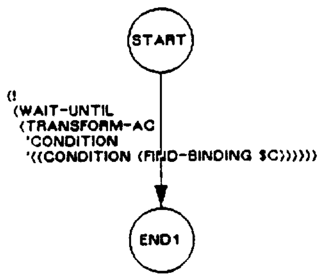
INVOCATION:
```
(AND (*GOAL (! (MONITOR $C)))
     (*FACT (FOO BAR)))
```

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL



switch-dilemma-2

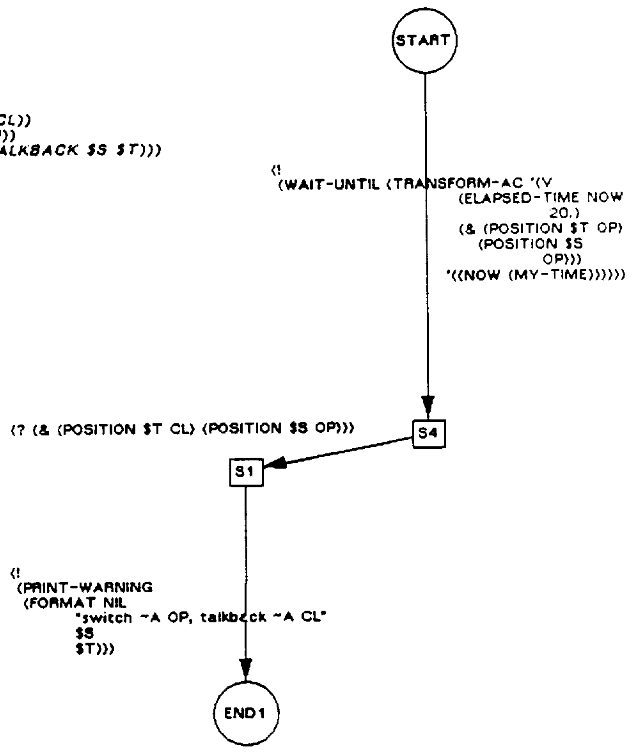
INVOCATION:
```
(AND (*FACT (POSITION $T CL))
     (*FACT (POSITION $S OP))
     (*FACT (ASSOCIATED-TALKBACK $S $T)))
```

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL



```
(!
 (WAIT-UNTIL (TRANSFORM-AC '(V
                          (ELAPSED-TIME NOW
                                        20.)
                          (& (POSITION $T OP)
                             (POSITION $S
                                       OP)))
                         '((NOW (MY-TIME))))))
```

```
(? (& (POSITION $T CL) (POSITION $S OP)))
```

```
(!
 (PRINT-WARNING
  (FORMAT NIL
          "switch ~A OP, talkback ~A CL"
          $S
          $T)))
```

retract

INVOCATION:
*(*GOAL (! (RETRACTED $P)))*

CONTEXT:
NIL

GOAL ACHIEVER?:
T

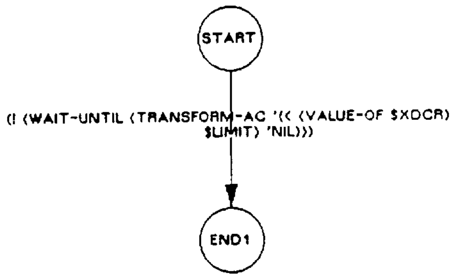
EFFECTS:
NIL

PROPERTIES:
NIL

ACTION:
(DELETE-FACT-FROM-KA $P)

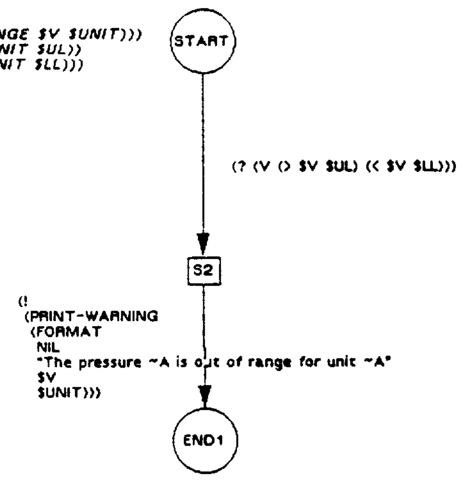
monitor-pressure

```
INVOCATION:
(AND (*GOAL (! (MONITOR (PRESSURE-BELOW $TK
                                       $LIMIT))))
     (*FACT (ASSOCIATED-UNIT $XDCR $TK))
     (*FACT (STATUS $XDCR GOOD)))

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL
```


START
(! (WAIT-UNTIL (TRANSFORM-AC '(< (VALUE-OF $XDCR) $LIMIT) 'NIL)))
END1

out-of-range

```
INVOCATION:
(AND (*GOAL (? (OUT-OF-RANGE $V $UNIT)))
     (*FACT (PRESSURE-UL $UNIT $UL))
     (*FACT (PRESSURE-LL $UNIT $LL)))

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL
```

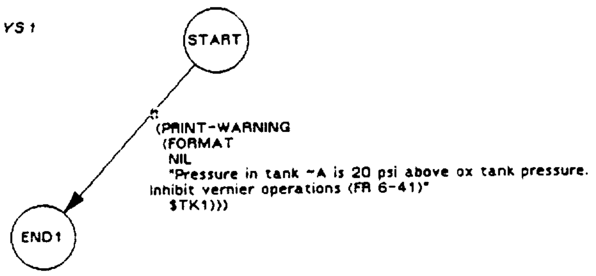


```
(? (V (> $V $UL) (< $V $LL)))
```

```
(!
 (PRINT-WARNING
  (FORMAT
   NIL
   "The pressure ~A is out of range for unit ~A"
   $V
   $UNIT)))
```

vernier-delta-p

```
INVOCATION:
(AND (*FACT (TYPE FUEL-TANK $TK1))
     (*FACT (PRESSURE $TK1 $P1))
     (*FACT (PART-OF $P-SYS1 $TK1))
     (*FACT (OTHER-PROPELLANT-SYSTEM $P-SYS1
                                     $P-SYS2))
     (*FACT (PART-OF $P-SYS2 $TK2))
     (*FACT (TYPE OXIDANT-TANK $TK2))
CON[illegible]T (PRESSURE $TK2 $P2))
NIL (*FACT (> (- $P1 $P2) 20.)))


GOAL ACHIEVER?:
T
```



```
(PRINT-WARNING
 (FORMAT
  NIL
  "Pressure in tank ~A is 20 psi above ox tank pressure.
Inhibit vernier operations (FR 6-41)"
  $TK1)))
```

```
EFFECTS:
NIL


PROPERTIES:
NIL
```

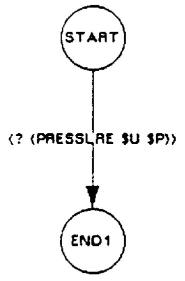
pressure-downstream

INVOCATION:
```
(AND (*GOAL (? (PRESSURE $TK $P)))
     (*FACT (CONNECTS $V $TK $U))
     (*FACT (POSITION $V OP)))
```

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL



PROPERTIES:
NIL

pressure-request

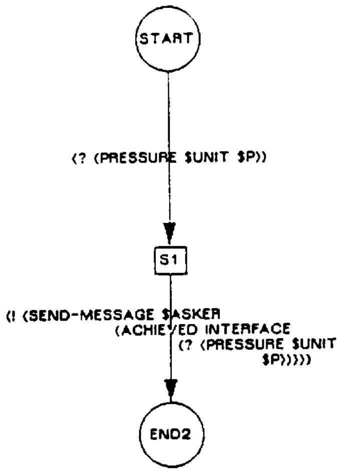
INVOCATION:
```
(*FACT (REQUEST $ASKER
          (? (PRESSURE $UNIT $P-ANY))))
```

CONTEXT:
NIL

GOAL ACHIEVER?:
T

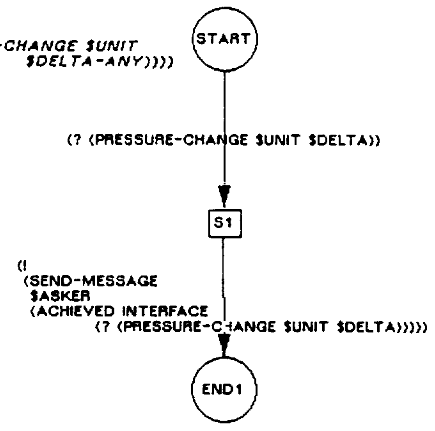
EFFECTS:
NIL

PROPERTIES:
NIL


START
(? (PRESSURE $UNIT $P))
S1
(! (SEND-MESSAGE $ASKER (ACHIEVED INTERFACE (? (PRESSURE $UNIT $P)))))
END2

request-pressure-change

```
INVOCATION:
(*FACT (REQUEST $ASKER
                (? (PRESSURE-CHANGE $UNIT
                                    $DELTA-ANY))))

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL
```


START
(? (PRESSURE-CHANGE $UNIT $DELTA))
S1
(! (SEND-MESSAGE $ASKER (ACHIEVED INTERFACE (? (PRESSURE-CHANGE $UNIT $DELTA)))))
END1

pressure-change

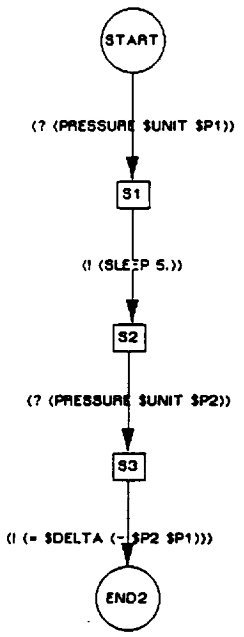
INVOCATION:
*(*GOAL (? (PRESSURE-CHANGE $UNIT $DELTA)))*

CONTEXT:
NIL

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL



selector2

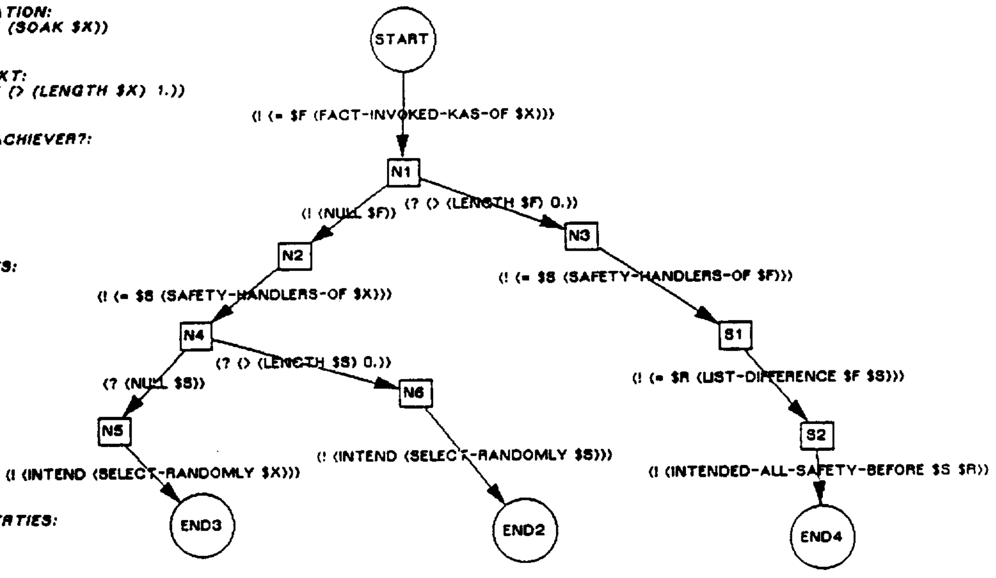
INVOCATION:
*(*FACT (SOAK $X))*

CONTEXT:
*(*FACT (> (LENGTH $X) 1.))*

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL


START
(! (= $F (FACT-INVOKED-KAS-OF $X)))
N1
(! (NULL $F))
(? (> (LENGTH $F) 0.))
N2
N3
(! (= $S (SAFETY-HANDLERS-OF $X)))
(! (= $S (SAFETY-HANDLERS-OF $F)))
N4
S1
(? (> (LENGTH $S) 0.))
(? (NULL $S))
N6
(! (= $R (LIST-DIFFERENCE $F $S)))
N5
S2
(! (INTEND (SELECT-RANDOMLY $S)))
(! (INTEND (SELECT-RANDOMLY $X)))
(! (INTENDED-ALL-SAFETY-BEFORE $S $R))
END3
END2
END4

selector

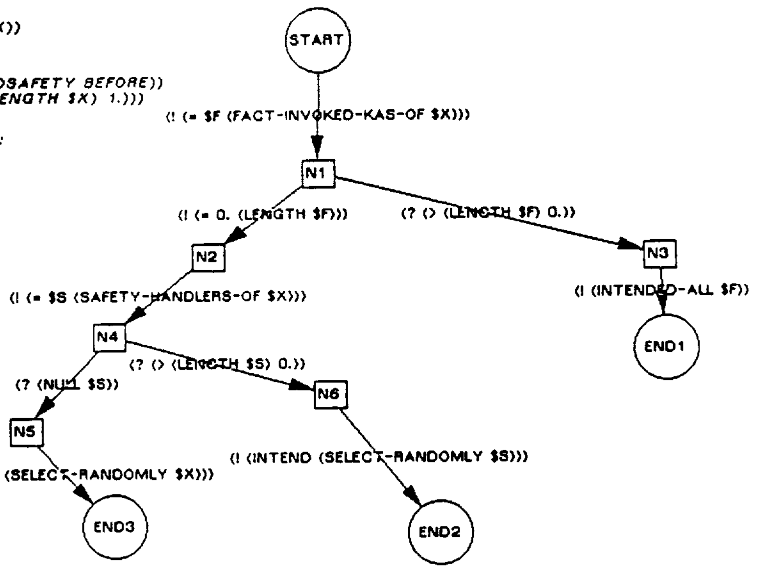
INVOCATION:
*(*FACT (SOAK $X))*

CONTEXT:
*(AND (*FACT (NOSAFETY BEFORE))*
*(*FACT (> (LENGTH $X) 1.)))*

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
NIL


START
(! (= $F (FACT-INVOKED-KAS-OF $X)))
N1
(! (= 0. (LENGTH $F)))
(? (> (LENGTH $F) 0.))
N2
N3
(! (= $S (SAFETY-HANDLERS-OF $X)))
(! (INTENDED-ALL $F))
N4
END1
(? (> (LENGTH $S) 0.))
(? (NULL $S))
N6
N5
(! (INTEND (SELECT-RANDOMLY $S)))
(! (INTEND (SELECT-RANDOMLY $X)))
END3
END2